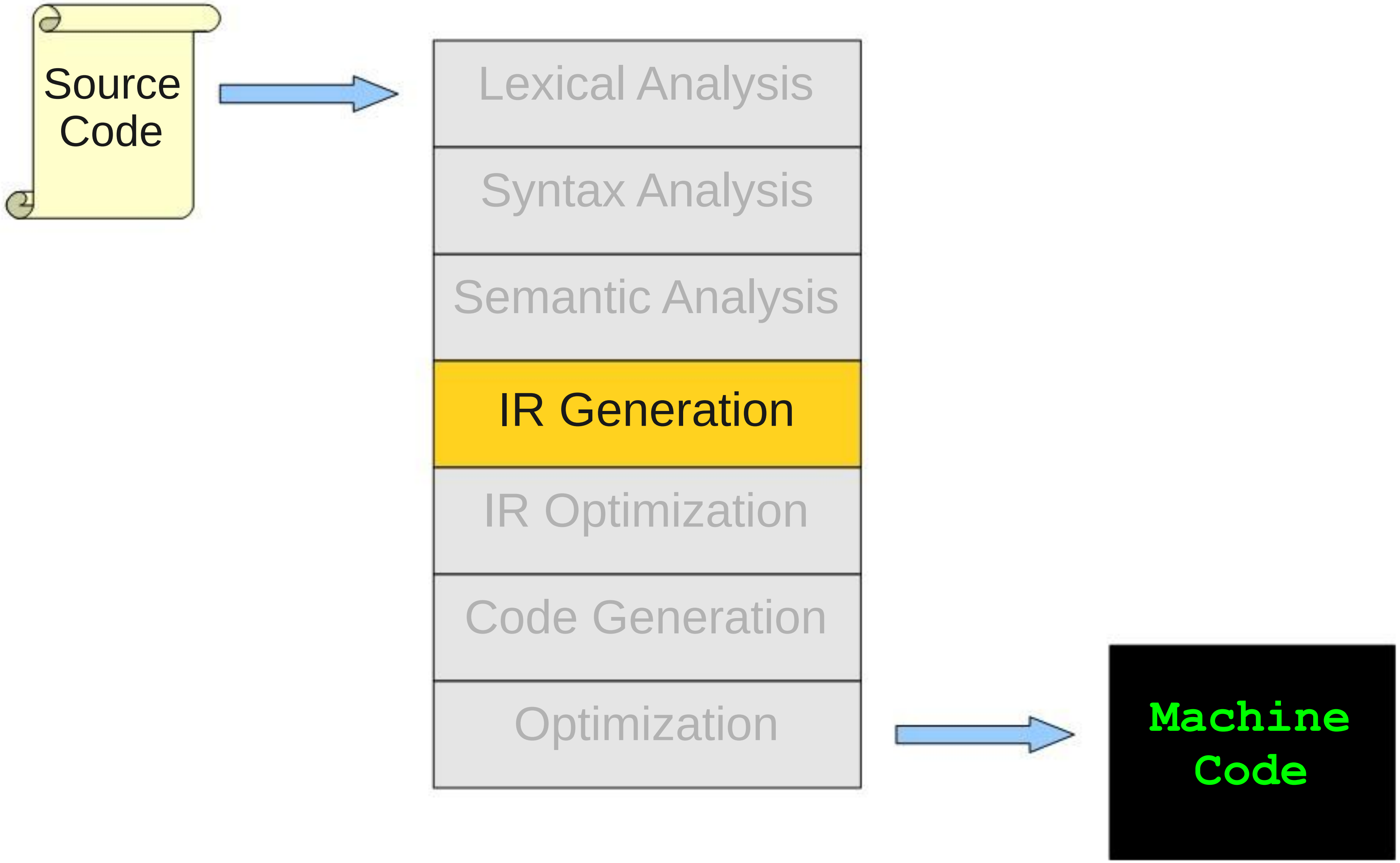
Three-Address Code IR

Where We Are


Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

Three-Address Code (TAC)

- High-level assembly language where each operation has at most three operands.
- Uses explicit runtime stack for function calls.
- When generating IR at this level, you do not need to worry about optimizing it.
- It's okay to generate IR that has lots of unnecessary assignments, redundant computations, etc.

Sample TAC Code

```
a = b + c + d;
b = a * a + b * b;
```

```
_t0 = b + c;
a = _t0 + d;
_t1 = a * a;
_t2 = b * b;
b = _t1 + _t2;
```

- Evaluating an expression sometimes requires the introduction of **temporary variables**.

Simple TAC Instructions

- **Variable assignment** allows assignments of the form
  - `var = constant;`
  - $\texttt{var}_1$ `=` $\texttt{var}_2$`;`
  - $\texttt{var}_1$ `=` $\texttt{var}_2$ **`op`** $\texttt{var}_3$`;`
  - $\texttt{var}_1$ `= constant` **`op`** $\texttt{var}_2$`;`
  - $\texttt{var}_1$ `=` $\texttt{var}_2$ **`op`** `constant;`
  - `var =` $\texttt{constant}_1$ **`op`** $\texttt{constant}_2$`;`
- Permitted operators are **`+`**, **`-`**, **`*`**, **`/`**, `%`
- How would you compile **`y = -x`**`;` ?

`y = 0 - x``;`

One More with `bool`s

```
int x;
int y;
bool b1;
bool b2;
bool b3;

b1 = x + x < y
b2 = x + x == y
b3 = x + x > y
```

```
_t0 = x + x;
_t1 = y;
b1 = _t0 < _t1;
```

```
_t2 = x + x;
_t3 = y;
b2 = _t2 == _t3;
```

```
_t4 = x + x;
_t5 = y;
b3 = _t5 < _t4;
```

‘>’ operator is not allowed.

TAC with `bool`s

- Boolean variables are represented as integers that have zero or nonzero values.

- In addition to the arithmetic operator, TAC supports **logical operators** `<`, `==`, `||`, and `&&`.

- How might you compile `b = (x <= y)` ?

```
_t0 = x < y;
_t1 = x == y;
b = _t0 || _t1;
```

Control Flow Statements

```
if (x < y)
    z = x;
else
    z = y;

z = z * z;
```

```
If x < y then _t0 = 1
Else _t0 = 0

    _t0 = x < y;
    IfZ _t0 Goto _L0;
    z = x;
    Goto _L1;
_L0:
    z = y;
_L1:
    z = z * z;
```

Labels

- TAC allows for **named labels** indicating particular points in the code that can be jumped to.
- There are two control flow instructions:
  - `Goto label;`
  - `IfZ value Goto label;`
- Note that **`IfZ`** is always paired with **`Goto`**.

A Sample Program

```
void main() {
    int x, y;
    int m2 = x * x + y * y;

    while (m2 > 5) {
        m2 = m2 - x;
    }
}
```

24 bytes for local variables and temporaries
(6 * 4-bytes = 24 bytes)

```
main:
    BeginFunc 24;
    _t0 = x * x;
    _t1 = y * y;
    m2 = _t0 + _t1;
_L0:
    _t2 = 5 < m2;
    IfZ _t2 Goto _L1;
    m2 = m2 - x;
    Goto _L0;
_L1:
    EndFunc;
```

Compiling Function Calls

Callee

```
void SimpleFn(int z) {
    int x, y;
    x = x * y * z;
}
```

Caller

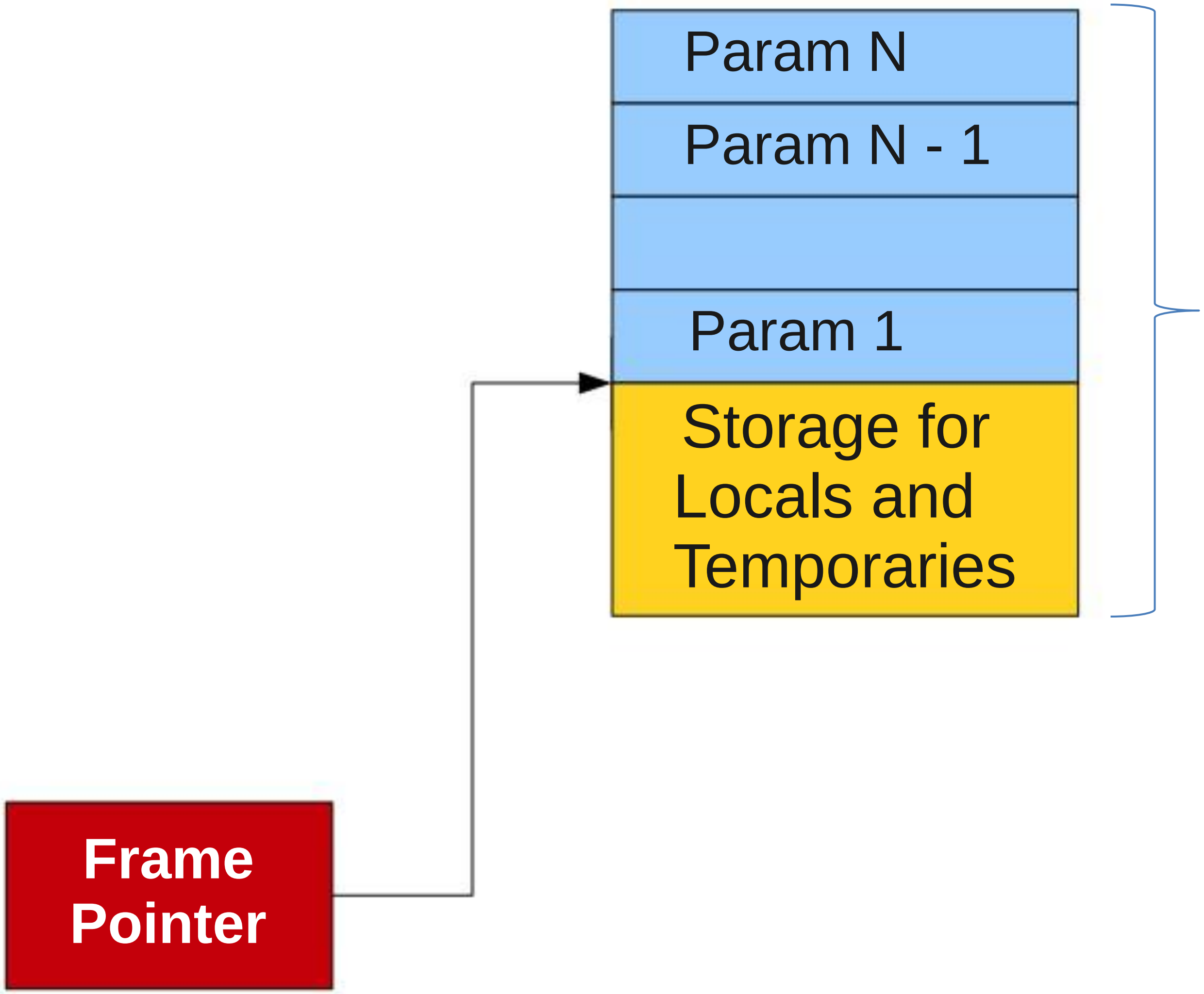
```
void main() {
    SimpleFn(137);
}
```

```
_SimpleFn:
    BeginFunc 16;
    _t0 = x * y;
    _t1 = _t0 * z;
    x = _t1;
    EndFunc;
```

Parameter z is already allocated by the caller

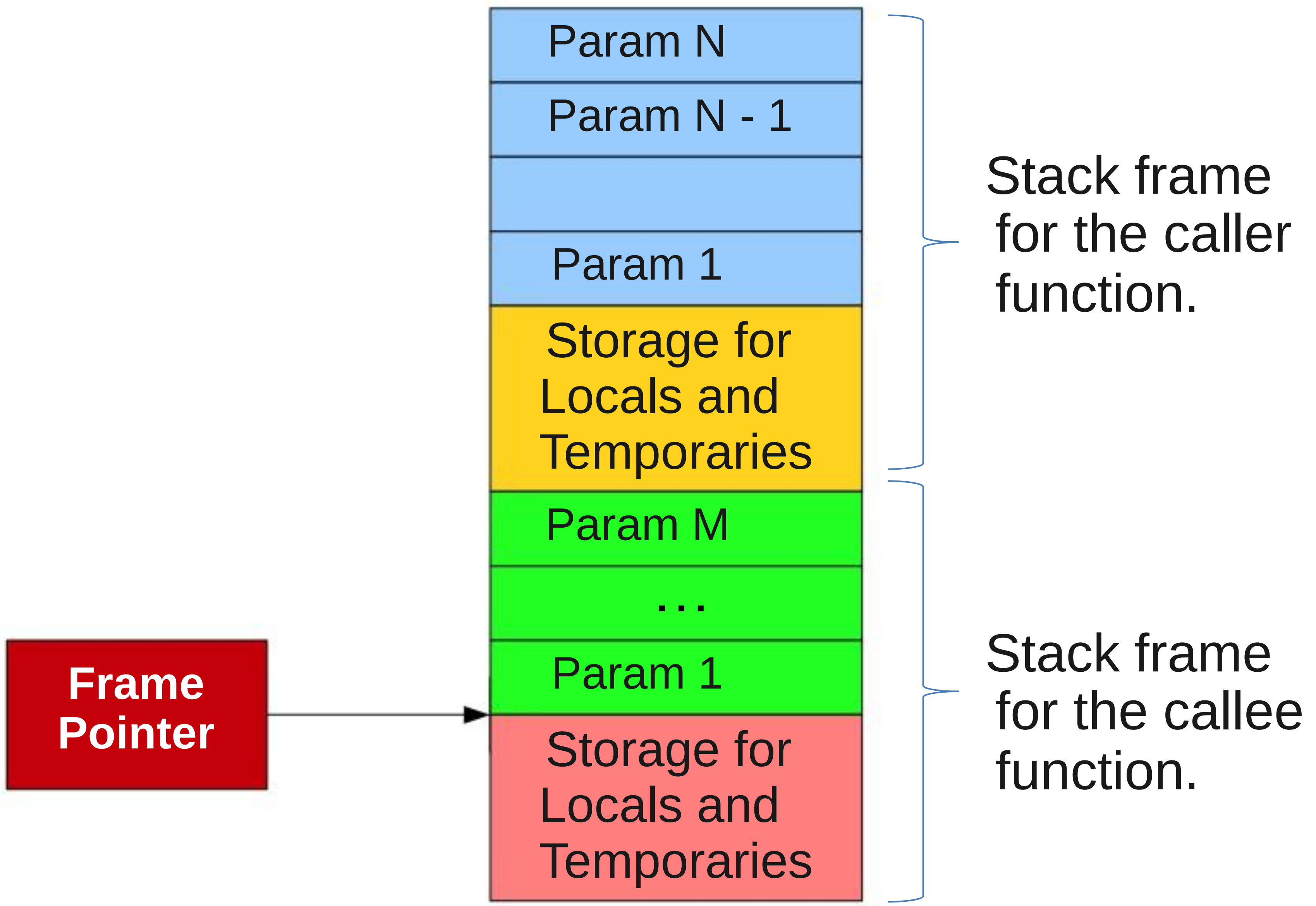
```
main:
    BeginFunc 4;
    _t0 = 137;
    PushParam _t0;
    LCall _SimpleFn;
    PopParams 4;
    EndFunc;
```

The Frame Pointer

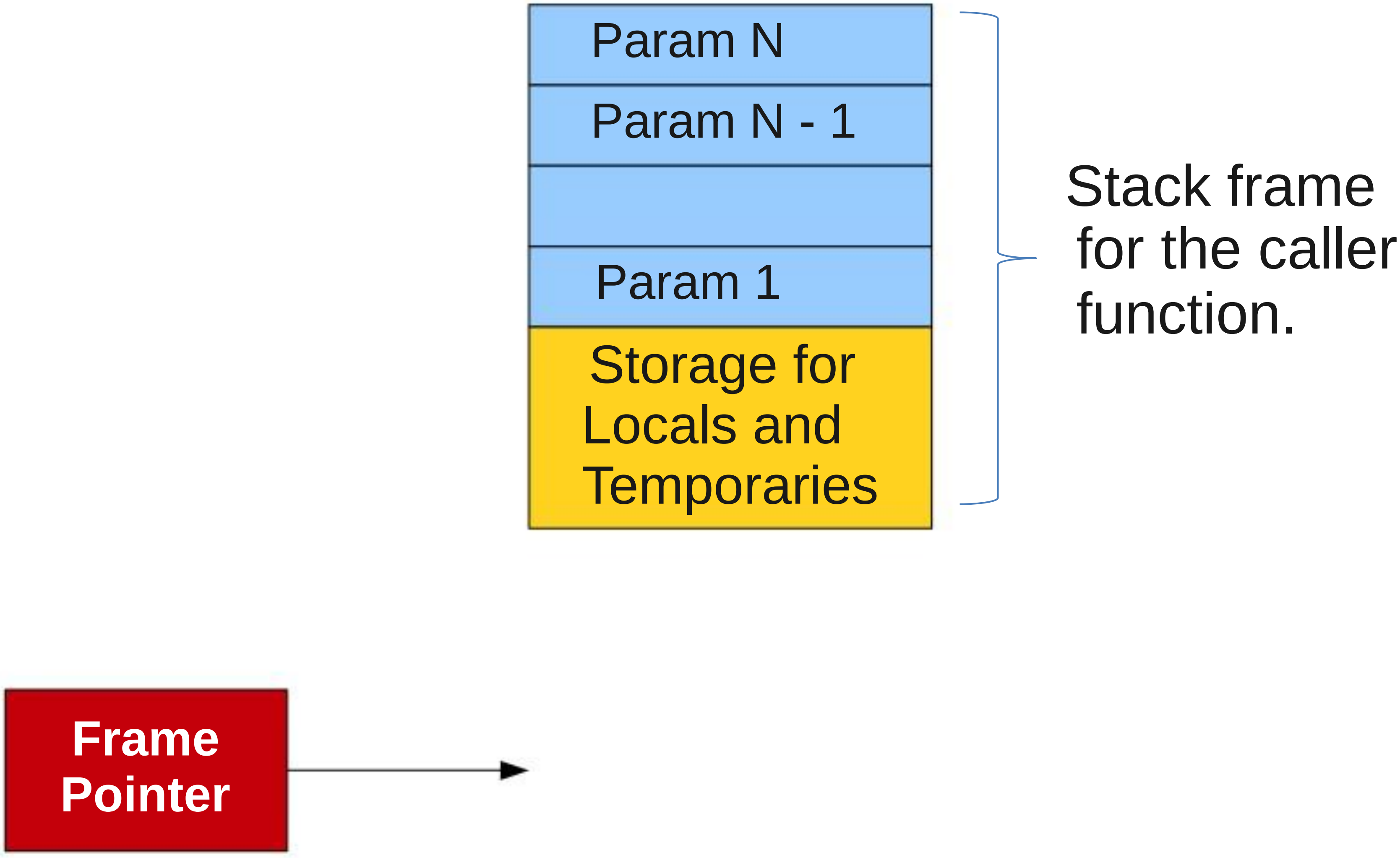


Stack frame for the caller function.

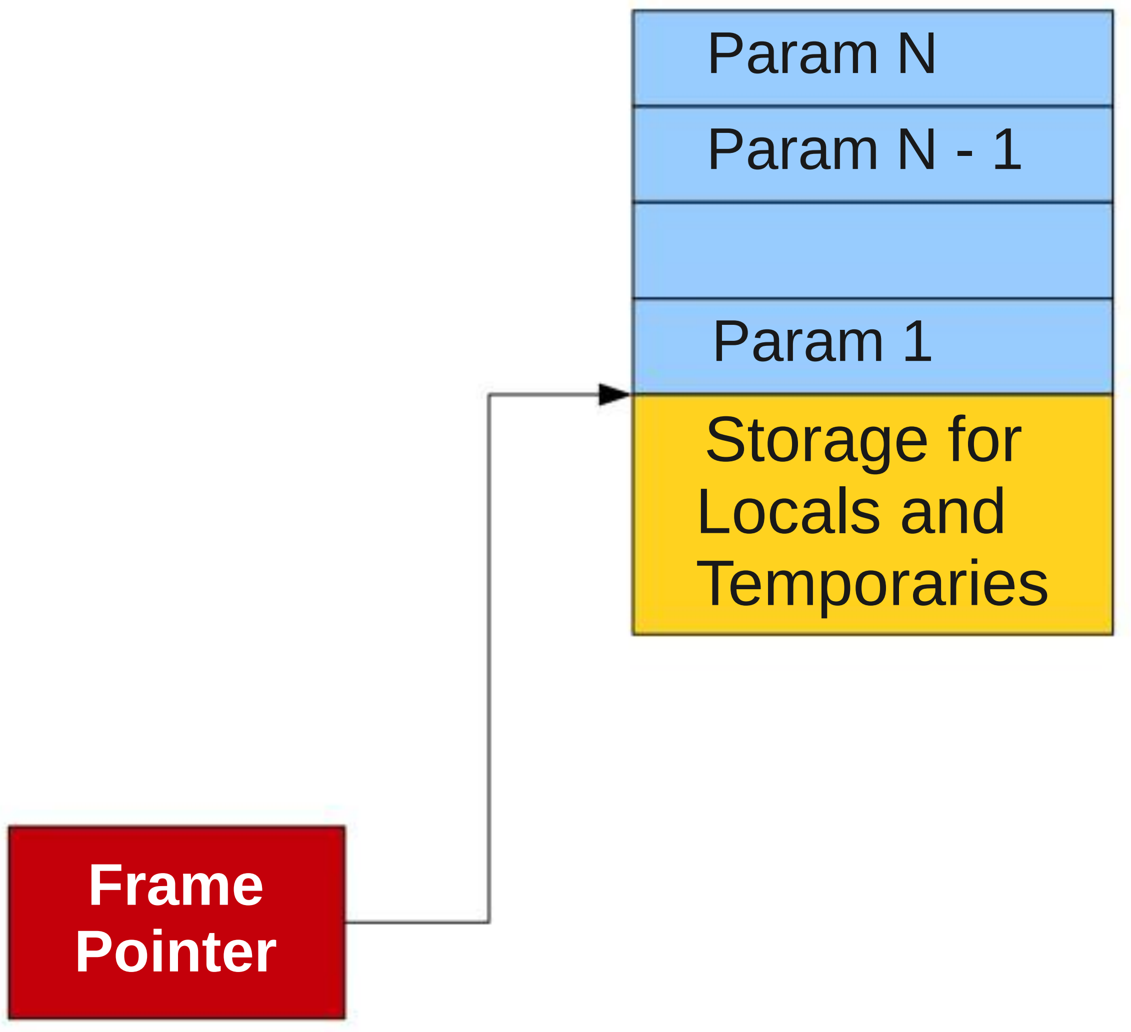
The Frame Pointer



The Frame Pointer

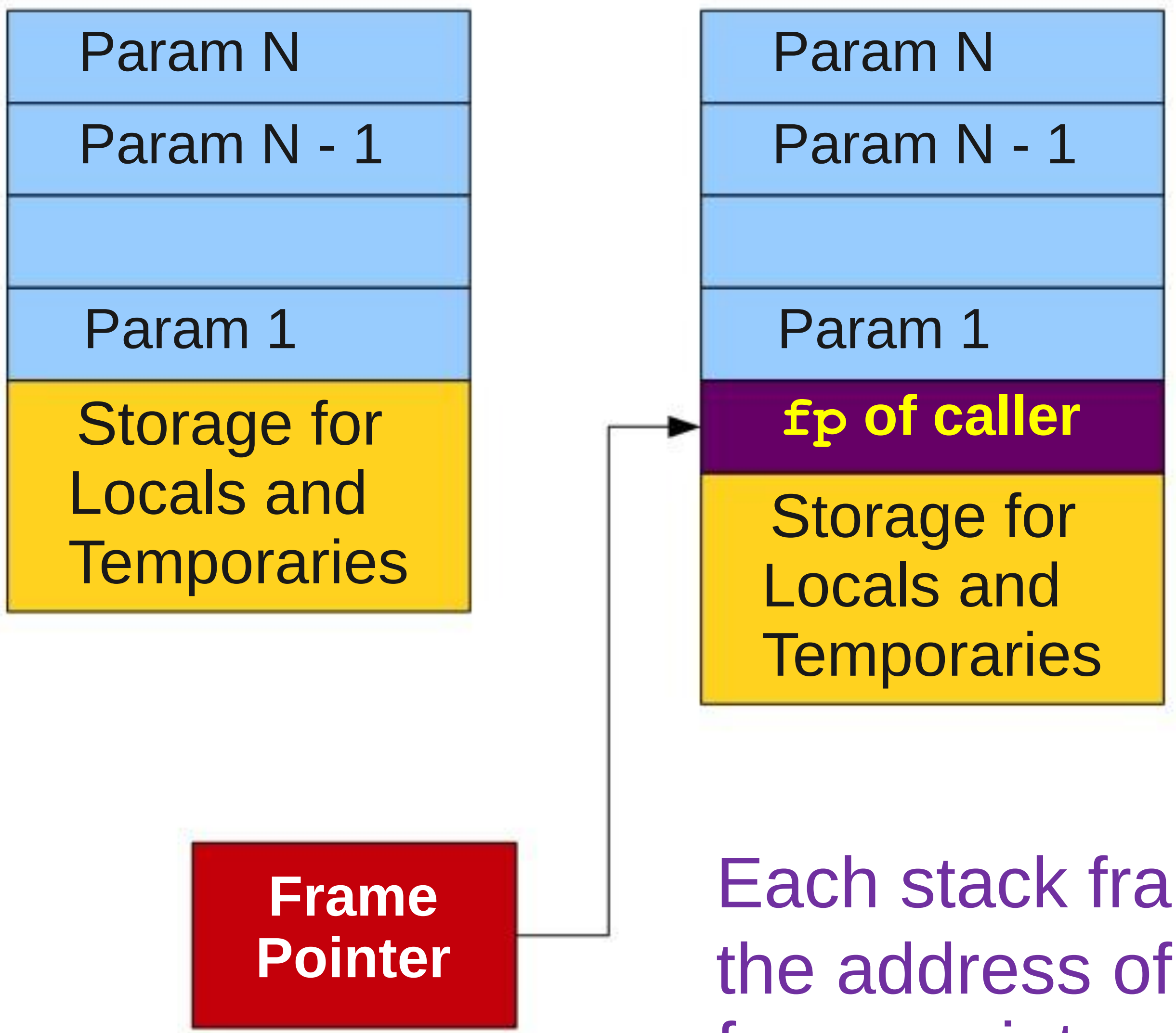


The Frame Pointer



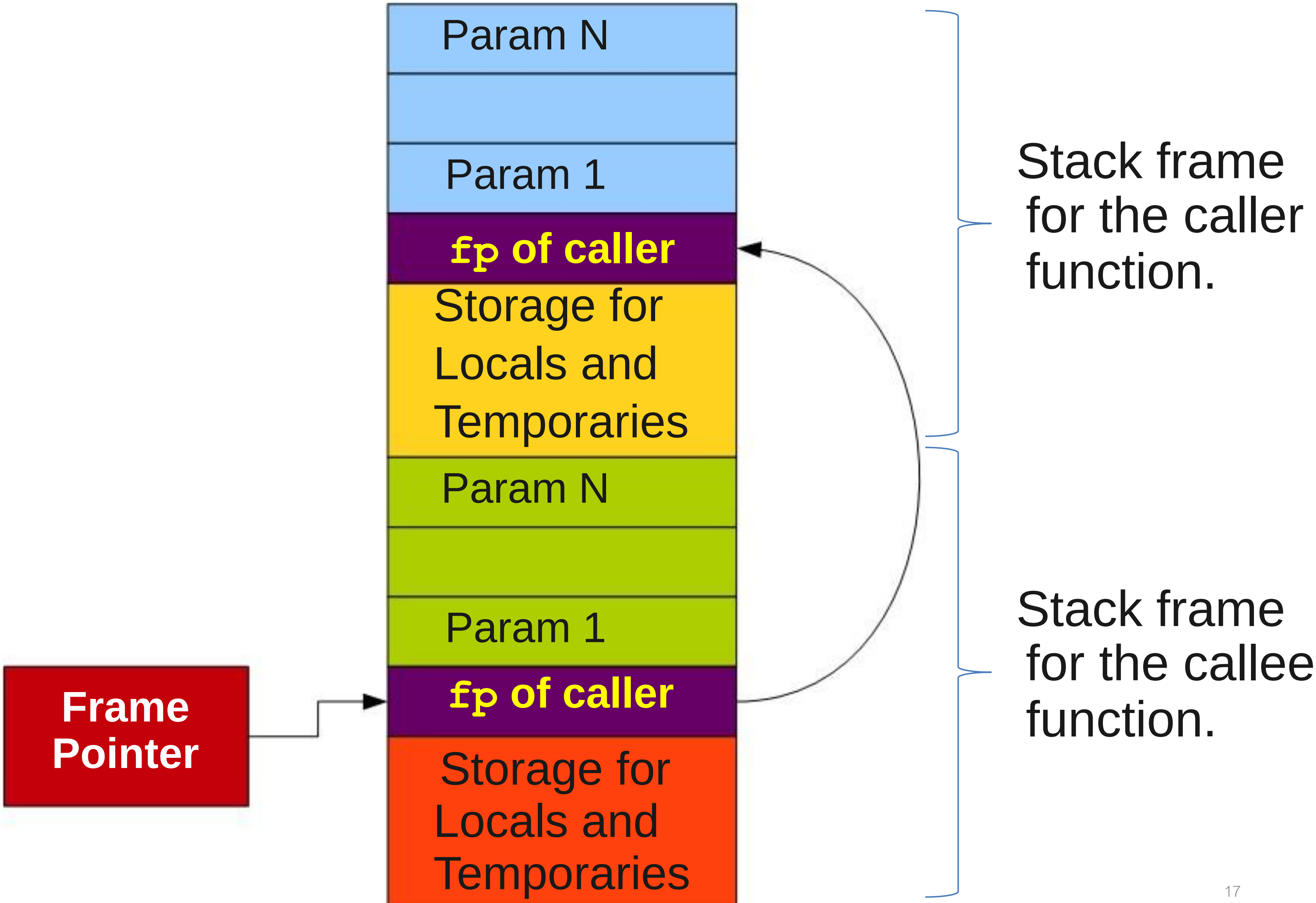
How do we know the position to move fp back?

Logical vs Physical Stack Frames

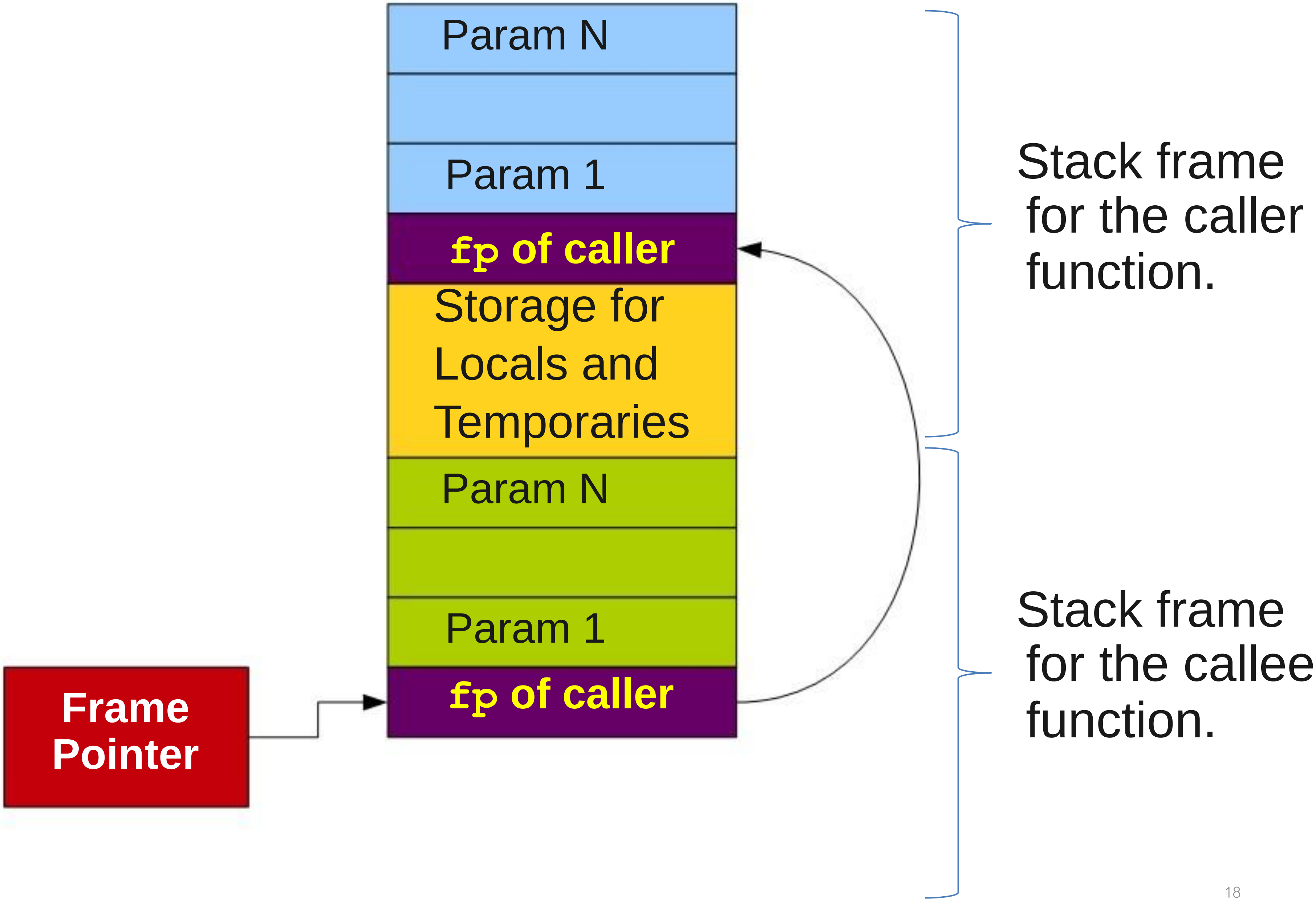


Each stack frame contains the address of the previous frame pointer.

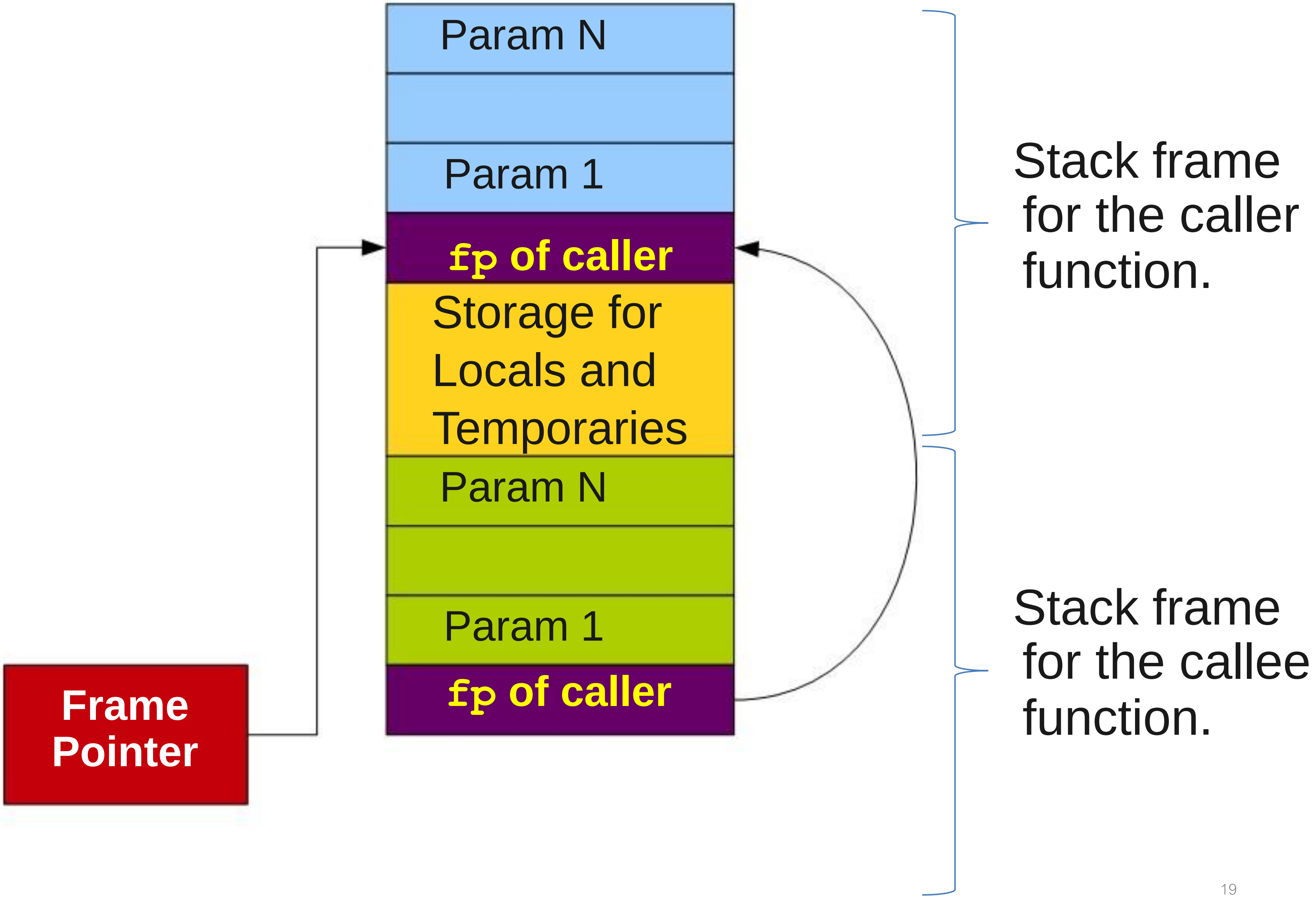
Physical Stack Frames



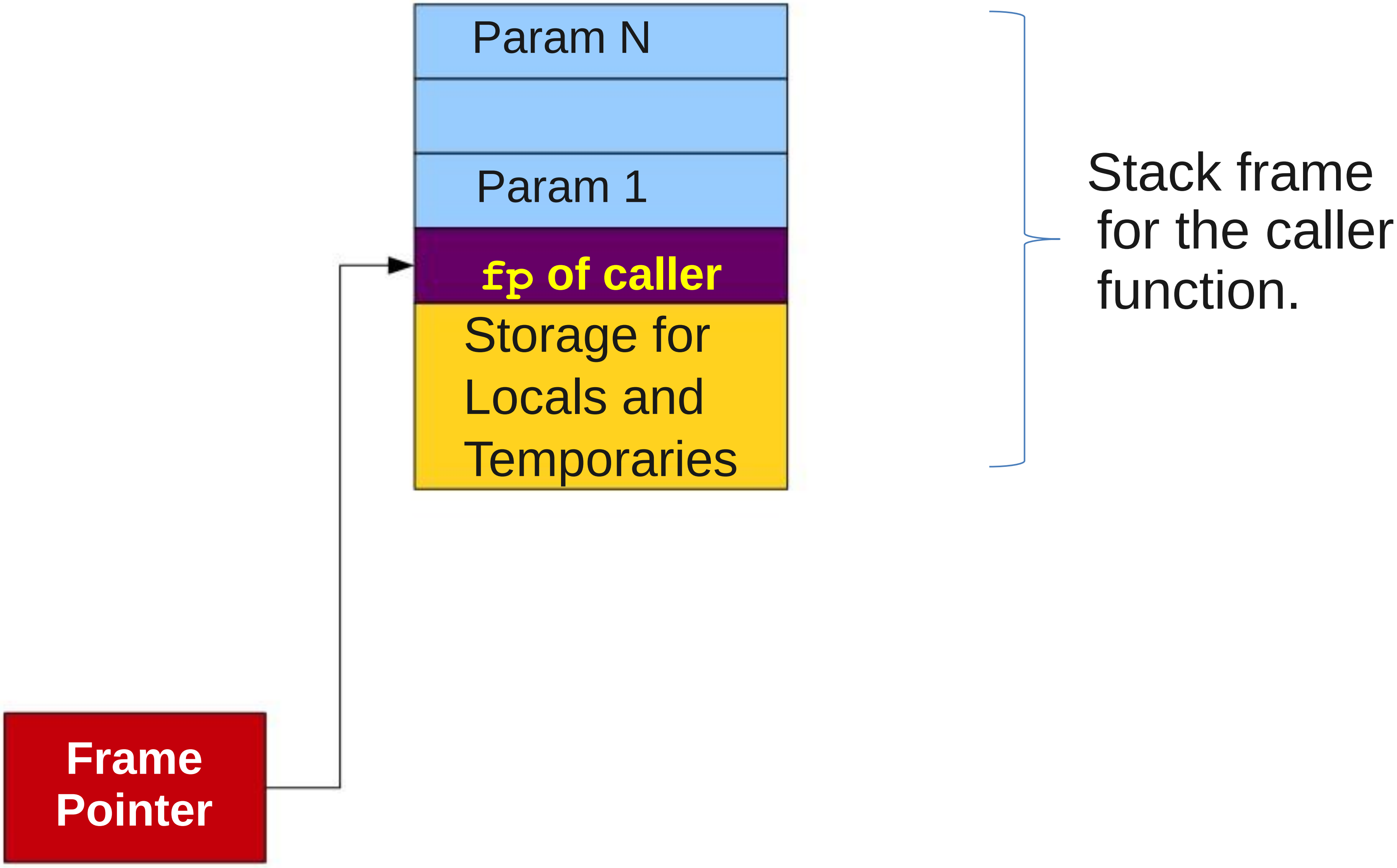
Physical Stack Frames



Physical Stack Frames



Physical Stack Frames



The Stored Return Address

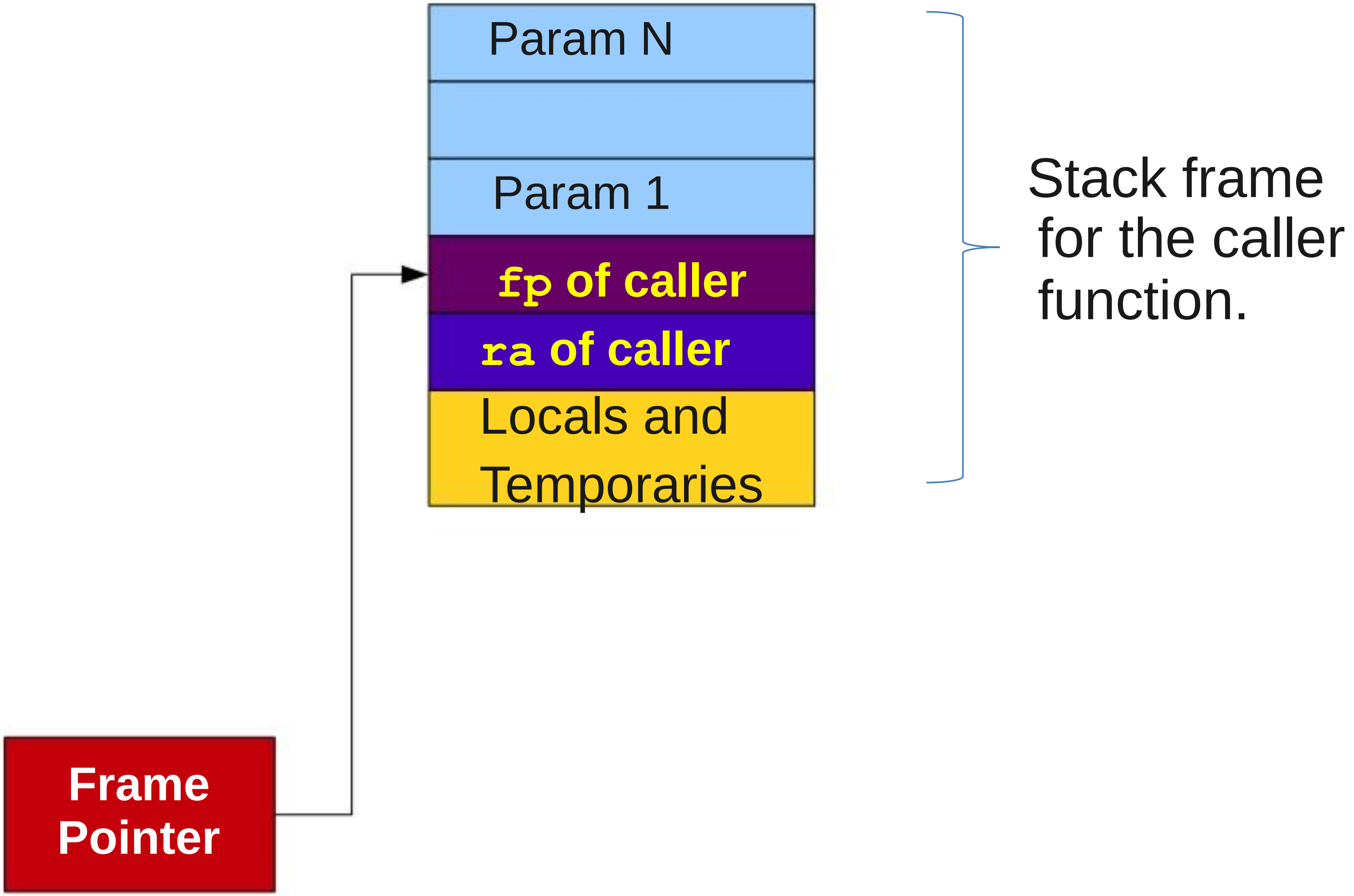
- Internally, the processor has a special register called the **program counter** (**PC**) that stores the address of the next instruction to execute.

- Whenever a function returns, it needs to restore the **PC** so that the calling function resumes execution where it left off.

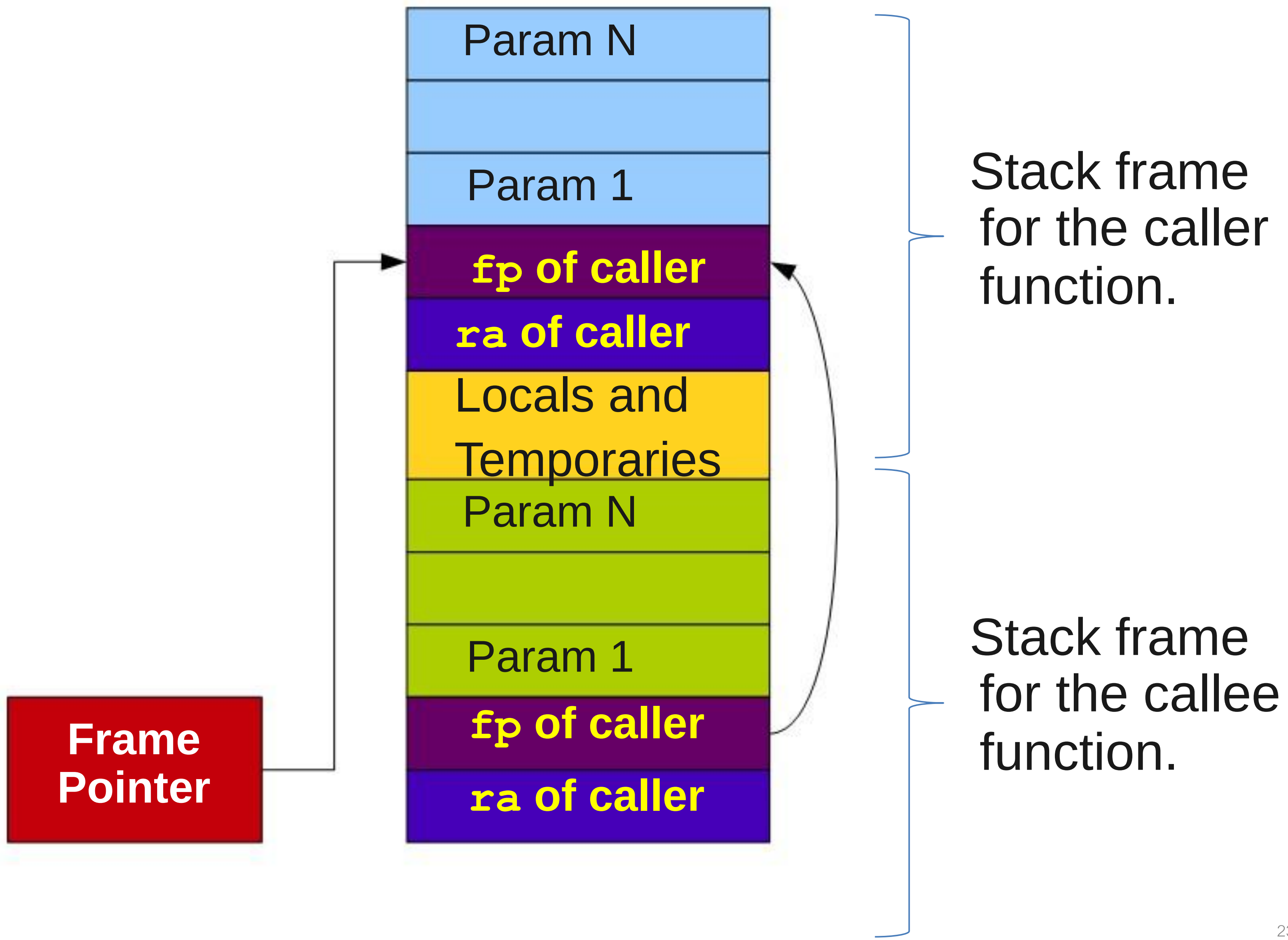
  E.g. The address of where to return is stored in **MIPS*** in a special register called **`ra`** ("return address.")

* **MIPS (Microprocessor without Interlocked Pipeline Stages)** is a reduced instruction set computer (RISC) instruction set (ISA) developed by MIPS Computer Systems, Inc. It compatible with both 32-Bit and 64-Bit architectures. It is widely used in small computing appliances, e.g. router, game station (PS2).

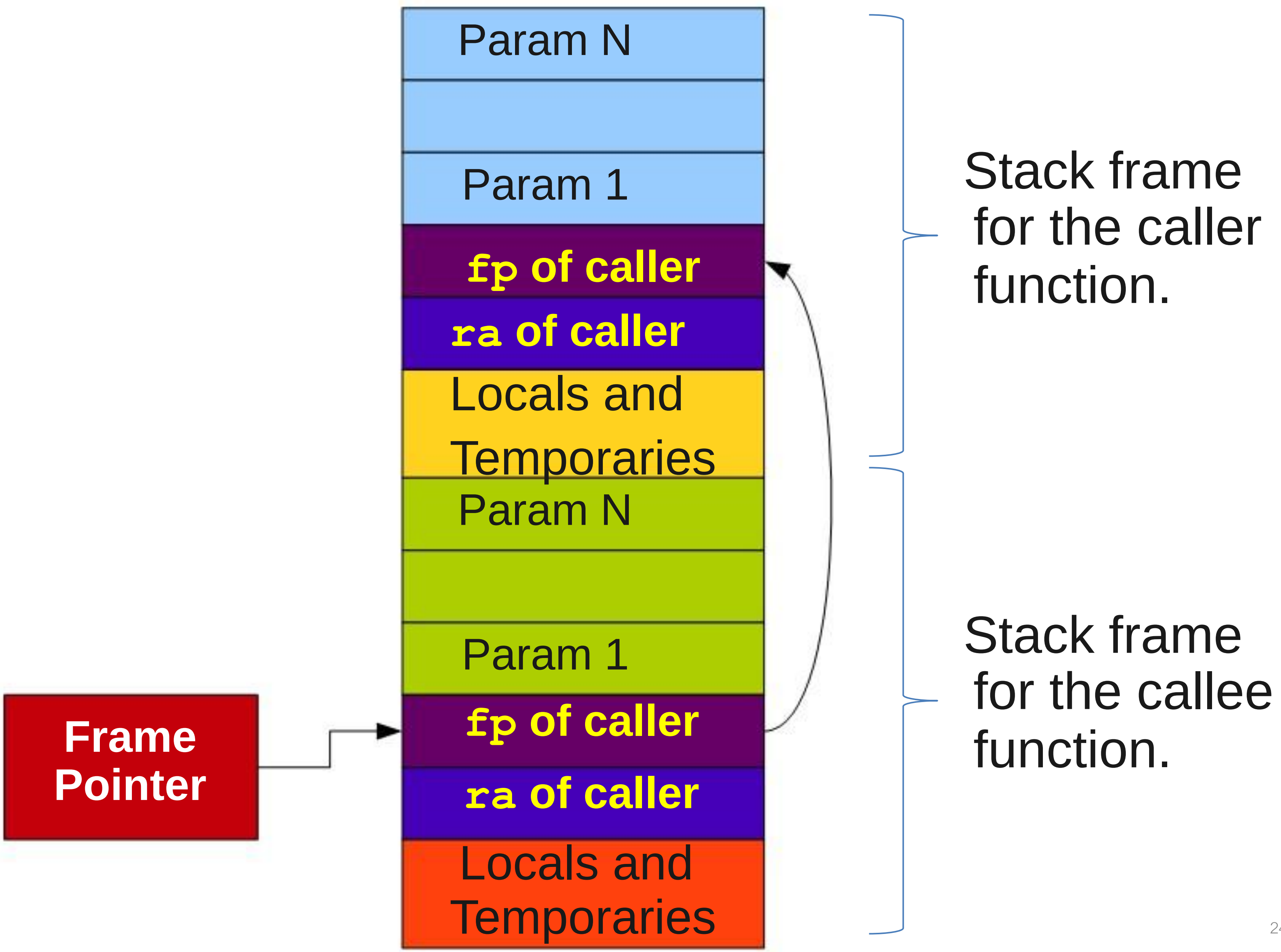
Physical Stack Frames



Physical Stack Frames



Physical Stack Frames

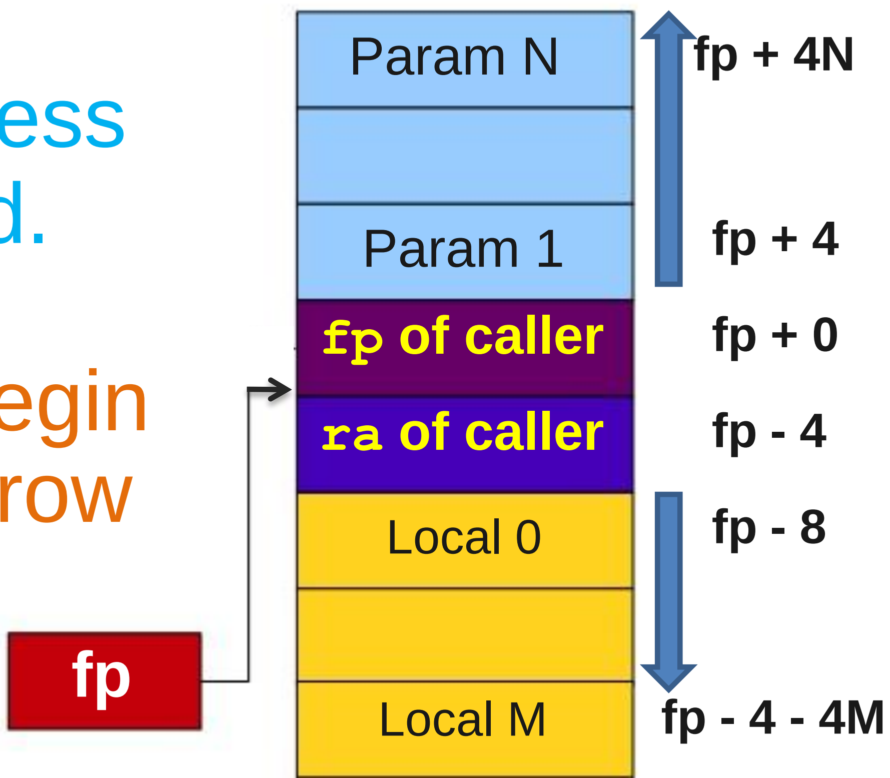


Locations of Parameters and Temp.

- Locations of local variable, parameter, and temporary variable are all relative to stack frame pointer (**fp-relative**).

- Parameters begin at address `fp + 4` and grow upward.

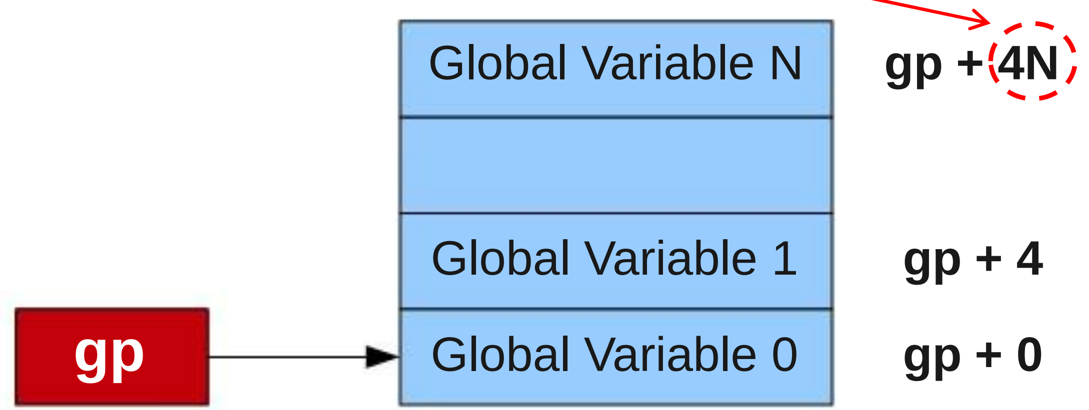
- Locals and temporaries begin at address `fp - 8` and grow downward.



The Global Pointer

- The **global pointer (gp)** points to the memory treated as an array of values that grows upward.

- You must choose an **offset** into this array for each global variable.



Generating TAC

cgen for Atomic Expressions

cgen(*c*) = {   // *c* is a constant
    Choose a new temporary *t*
    Emit(*t* = *c*);
    Return *t*
}

cgen(*id*) = {   // *id* is an identifier
    Choose a new temporary *t*
    Emit(*t* = *id*)
    Return *t*
}

cgen for Binary Operators

cgen($e_1 + e_2$) = {

Choose a new temporary t

Let t_1 = **cgen**(e_1)

Let t_2 = **cgen**(e_2)

Emit($t = t_1 + t_2$)

Return t

}

An Example

cgen(5 + x) = {

Choose a new temporary *t*

Let t_1 = **cgen**(5)

Let t_2 = **cgen**(x)

Emit ($t = t_1 + t_2$)

Return t

}

An Example

cgen(5 + x) = {
    Choose a new temporary *t*
    Let t_1 = {
        Choose a new temporary *t*
        Emit(*t* = 5)
        return *t*
    }
    Let t_2 = **cgen**(x)
    Emit (t = t_1 + t_2)
    Return t
}

```
_t1 = 5
```

An Example

cgen(5 + x) = {
  Choose a new temporary *t*
  Let t_1 = {
    Choose a new temporary *t*
    Emit(*t* = 5)
    return *t*
  }
  Let t_2 = {
    Choose a new temporary *t*
    Emit(*t* = x)
    return *t*
  }
  Emit (t = t_1 + t_2)
  Return t
}

```
_t1 = 5
_t2 = x
```

An Example

cgen(5 + x) = {
    Choose a new temporary *t*
    Let t_1 = {
        Choose a new temporary *t*
        Emit(*t* = 5)
        return *t*
    }
    Let t_2 = {
        Choose a new temporary *t*
        Emit(*t* = x)
        return *t*
    }
    Emit (*t* = t_1 + t_2)
    Return t
}

```
_t1 = 5
_t2 = x
_t3 = _t1 + _t2
```

cgen for Simple Statements

cgen(*expr* ***;***) = {

    cgen(*expr*)

}

cgen for `while` loops

cgen(**`while`** **`(`***expr***`)`** *stmt*) = {

}

cgen for **`while`** loops

cgen(**`while (`***expr***`)`** *stmt*) = {

Let L_{before} be a new label.

Let L_{after} be a new label.

}

cgen for **`while`** loops

cgen(**`while (`***expr***`)`** *stmt*) = {

Let L_{before} be a new label.

Let L_{after} be a new label.

Emit(L_{before} :)

Emit(L_{after} :)

}

cgen for `while` loops

cgen(`while (`*expr*`)` *stmt*) = {

Let L_{before} be a new label.

Let L_{after} be a new label.

Emit(L_{before} `:`)

Let *t* = **cgen**(*expr*)
Emit(`IfZ` *t* `Goto` L_{after})

Emit(L_{after} `:`)

}

cgen for **`while`** loops

cgen(**`while (`***expr***`)`** *stmt*) = {

Let L_{before} be a new label.

Let L_{after} be a new label.

Emit(L_{before} **`:`**)

Let *t* = **cgen**(*expr*)
Emit(**`IfZ`** *t* **`Goto`** L_{after})

cgen(*stmt*)

Emit(L_{after} **`:`**)

}

cgen for `while` loops

cgen(`while (`*expr*`)` *stmt*) = {

Let L_{before} be a new label.

Let L_{after} be a new label.

Emit(L_{before} **:**)

Let *t* = **cgen**(*expr*)
Emit(`IfZ` *t* `Goto` L_{after})

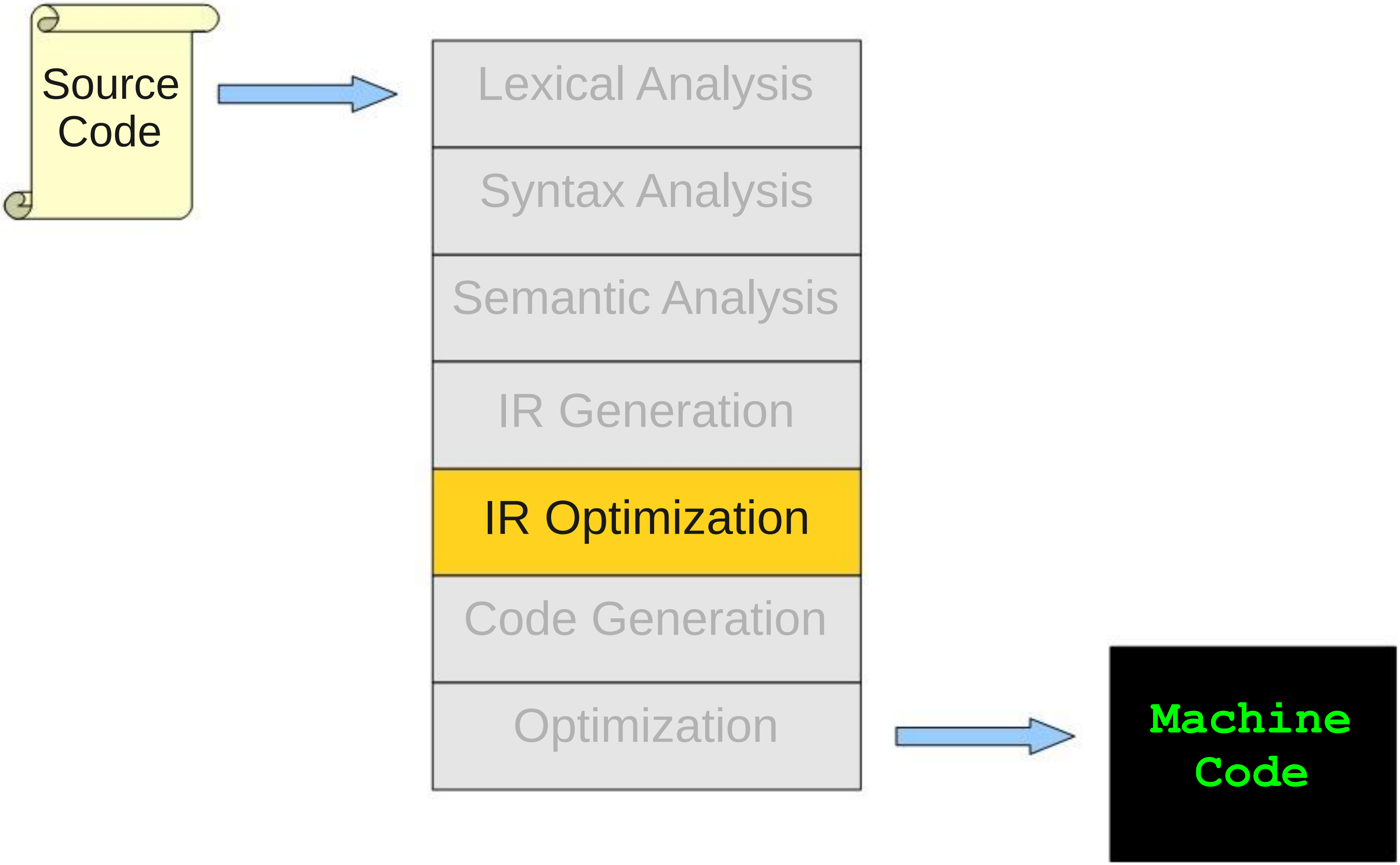
cgen(*stmt*)
Emit(`Goto` L_{before})

Emit(L_{after} **:**)

}

IR Optimization

Where We Are


Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

Optimizations from IR Generation

```
int x;
int y;
bool b1;
bool b2;
bool b3;

b1 = x + x < y
b2 = x + x == y
b3 = x + x > y
```

```
_t0 = x + x;
_t1 = y;
b1 = _t0 < _t1;

_t2 = x + x;
_t3 = y;
b2 = _t2 == _t3;

_t4 = x + x;
_t5 = y;
b3 = _t5 < _t4;
```

Can you see redundancies?

Optimizations from IR Generation

```
int x;
int y;
bool b1;
bool b2;
bool b3;

b1 = x + x < y
b2 = x + x == y
b3 = x + x > y
```

```
_t0 = x + x;
_t1 = y;
b1 = _t0 < _t1;

_t2 = x + x;
_t3 = y;
b2 = _t2 == _t3;

_t4 = x + x;
_t5 = y;
b3 = _t5 < _t4;
```

Optimizations from IR Generation

```
int x;
int y;
bool b1;
bool b2;
bool b3;

b1 = x + x < y
b2 = x + x == y
b3 = x + x > y
```

```
_t0 = x + x;
_t1 = y;
b1 = _t0 < _t1;



b2 = _t0 == _t1;



b3 = _t0 < _t1;
```

Optimizations from Lazy Coders

```
while (x < y + z) {
    x = x - y;
}
```

```
_L0:
    _t0 = y + z;
    _t1 = x < _t0;
    IfZ _t1 Goto _L1;
    x = x - y;
    Goto _L0;
_L1:
```

Optimizations from Lazy Coders

```
while (x < y + z) {
    x = x - y;
}
```

This code never changes.

```
_L0:
    _t0 = y + z;
    _t1 = x < _t0;
    IfZ _t1 Goto _L1;
    x = x - y;
    Goto _L0;
_L1:
```

Optimizations from Lazy Coders

```
while (x < y + z) {
    x = x - y;
}
```

```
    _t0 = y + z;
_L0:
    _t1 = x < _t0;
    IfZ _t1 Goto _L1;
    x = x - y;
    Goto _L0;
_L1:
```

The Challenge of Optimization

- **A good optimizer**
  - Should never change the observable behavior of a program.
  - Should produce IR that is as efficient as possible.
  - Should not take too long to process inputs.

What are we Optimizing?

What are some quantities we might want to optimize?

- **Runtime** (make the program as fast as possible at the expense of time and power)
- **Memory usage** (generate the smallest possible executable at the expense of time and power)
- **Power consumption** (choose simple instructions at the expense of speed and memory usage)
- Plus a lot more (minimize function calls, reduce use of floating-point hardware, etc.)

Semantics-Preserving Optimizations

- An optimization is **semantics-preserving** if it does not alter the semantics of the original program.
- Examples:
  - Eliminating unnecessary temporary variables.
  - Computing values that are known statically at compile-time instead of runtime.
  - Evaluating constant expressions outside of a loop instead of inside.

A Formalism for IR Optimization

- Every phase of the compiler uses some tools:
  - Scanning (Lexer) uses regular expressions.
  - Parsing uses CFGs.
  - Semantic analysis uses proof systems and symbol tables.

- In optimization, we need a **Control-Flow Graph** to capture the structure of a program in a way amenable to optimization.

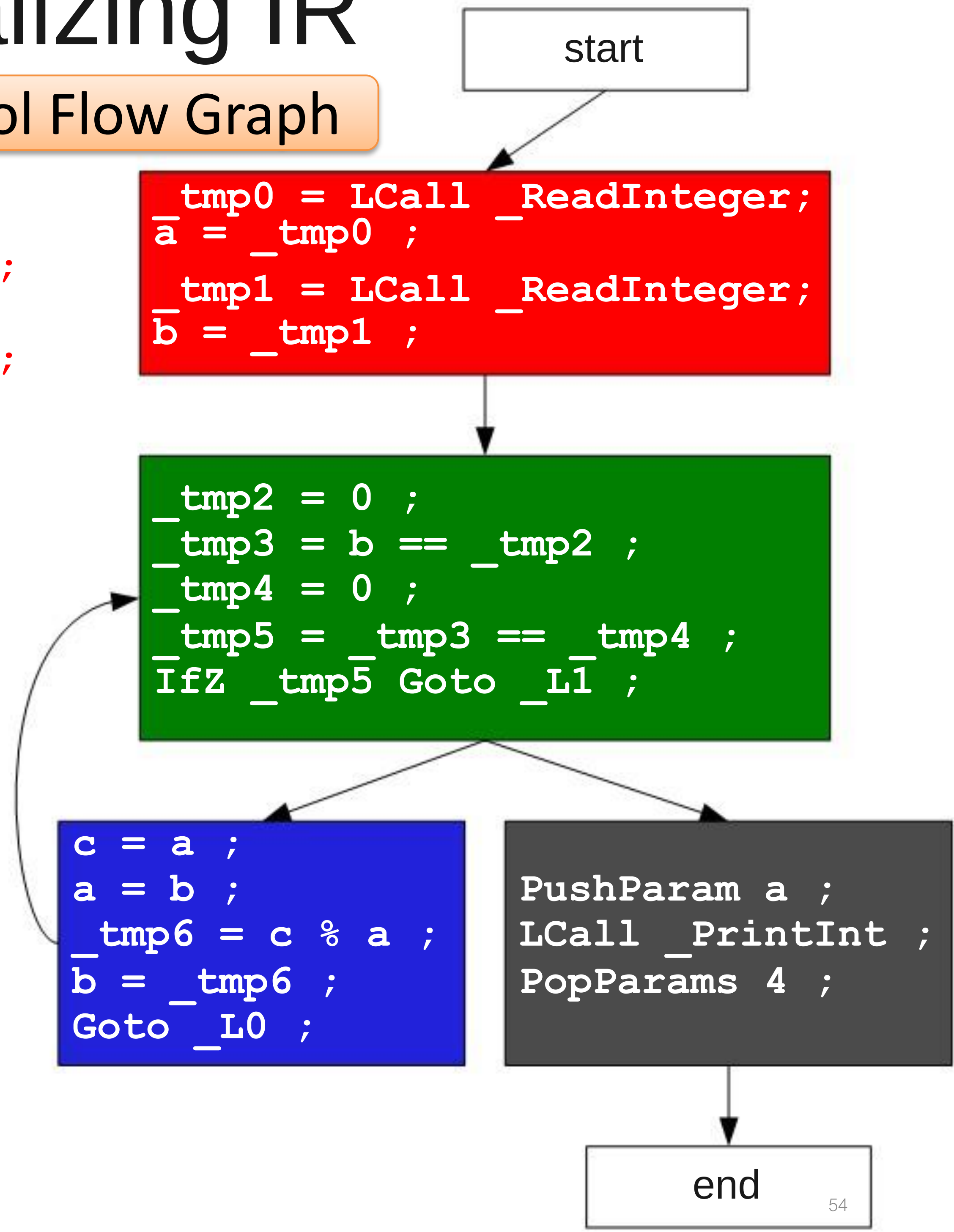
Visualizing IR

```
main:
    BeginFunc 40;
    _tmp0 = LCall _ReadInteger;
    a = _tmp0;
    _tmp1 = LCall _ReadInteger;
    b = _tmp1;
_L0:
    _tmp2 = 0;
    _tmp3 = b == _tmp2;
    _tmp4 = 0;
    _tmp5 = _tmp3 == _tmp4;
    IfZ _tmp5 Goto _L1;
    c = a;
    a = b;
    _tmp6 = c % a;
    b = _tmp6;
    Goto _L0;
_L1:
    PushParam a;
    LCall _PrintInt;
    PopParams 4;
    EndFunc;
```

Visualizing IR

Control Flow Graph

```
main:
    BeginFunc 40;
    _tmp0 = LCall _ReadInteger;
    a = _tmp0;
    _tmp1 = LCall _ReadInteger;
    b = _tmp1;
_L0:
    _tmp2 = 0;
    _tmp3 = b == _tmp2;
    _tmp4 = 0;
    _tmp5 = _tmp3 == _tmp4;
    IfZ _tmp5 Goto _L1;
    c = a;
    a = b;
    _tmp6 = c % a;
    b = _tmp6;
    Goto _L0;
_L1:
    PushParam a;
    LCall _PrintInt;
    PopParams 4;
    EndFunc;
```



Basic Blocks & Control-Flow Graph

- A **basic block** is a sequence of IR instructions where
  - There is exactly one spot where control enters the sequence, which must be at the start of the sequence.
  - There is exactly one spot where control leaves the sequence, which must be at the end of the sequence.
- A **control-flow graph** (CFG) is a graph of the basic blocks in a function.
  - The term CFG is overloaded - from here on out, we'll mean “control-flow graph” and not “context-free grammar.”

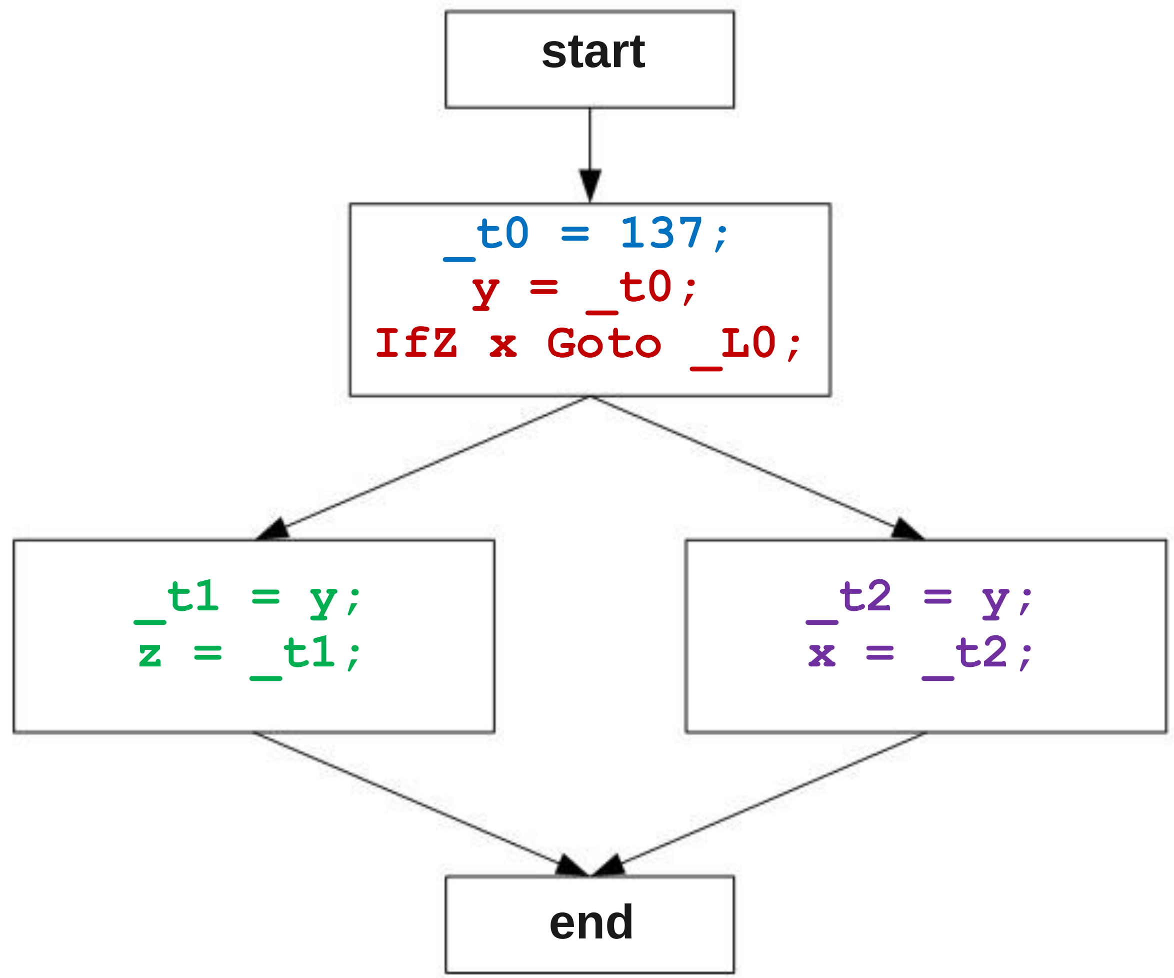
Types of Optimizations

- An optimization is **local** if it works on just a single basic block.
- An optimization is **global** if it works on an entire control-flow graph.
- An optimization is **interprocedural** if it works across the control-flow graphs of multiple functions. (We won't talk about this in this course.)

Example

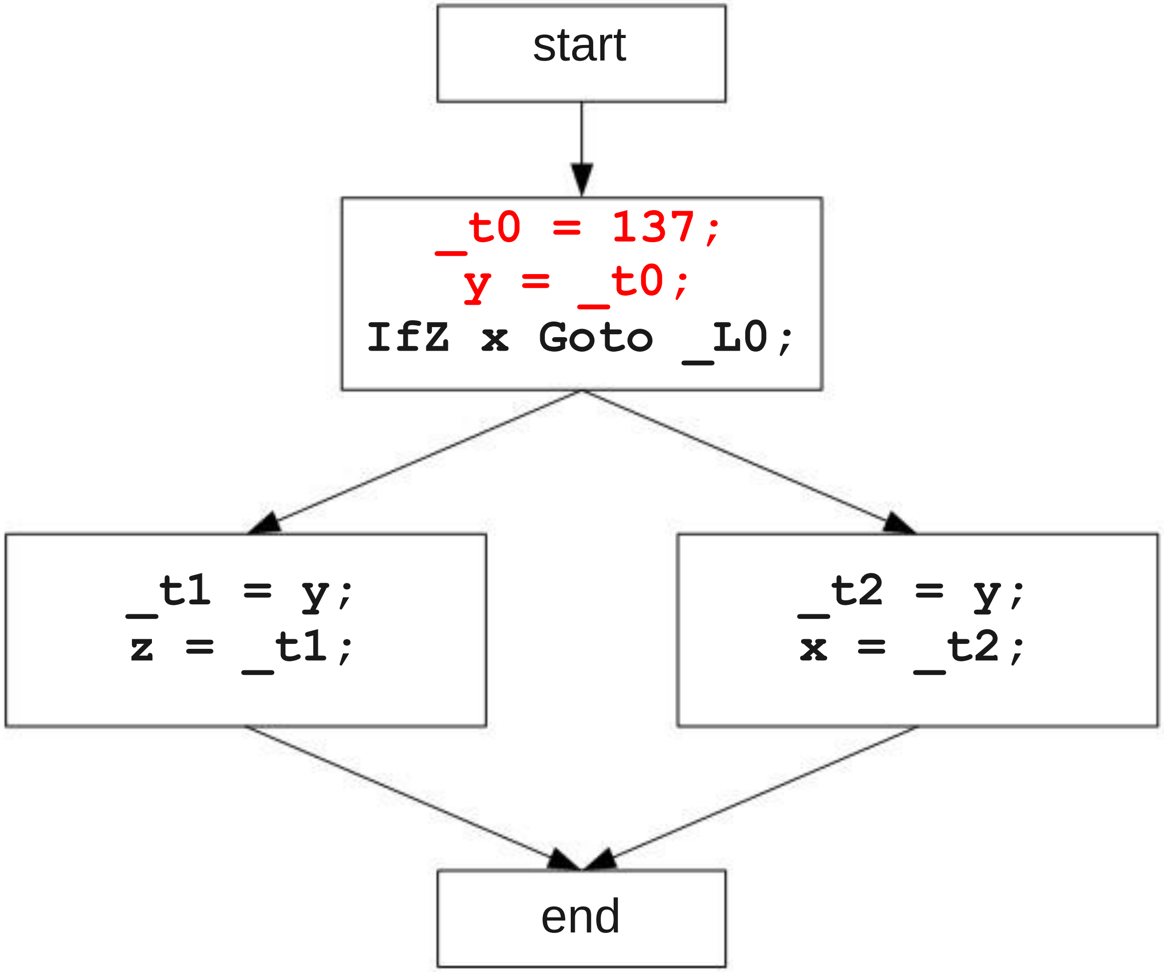
Local Optimizations

```
int main() {
    int x;
    int y;
    int z;

    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}
```



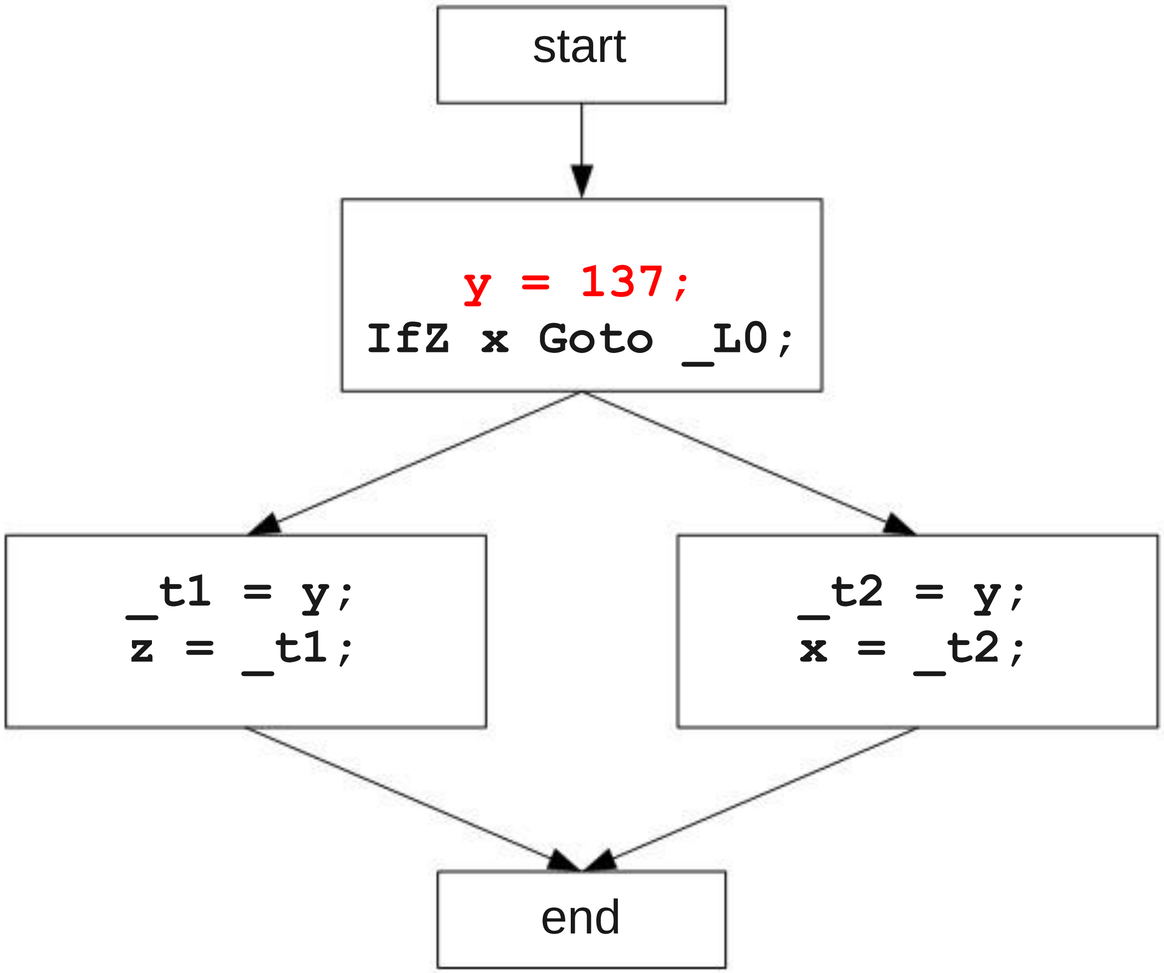
Local Optimizations

```
int main() {
    int x;
    int y;
    int z;

    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}
```



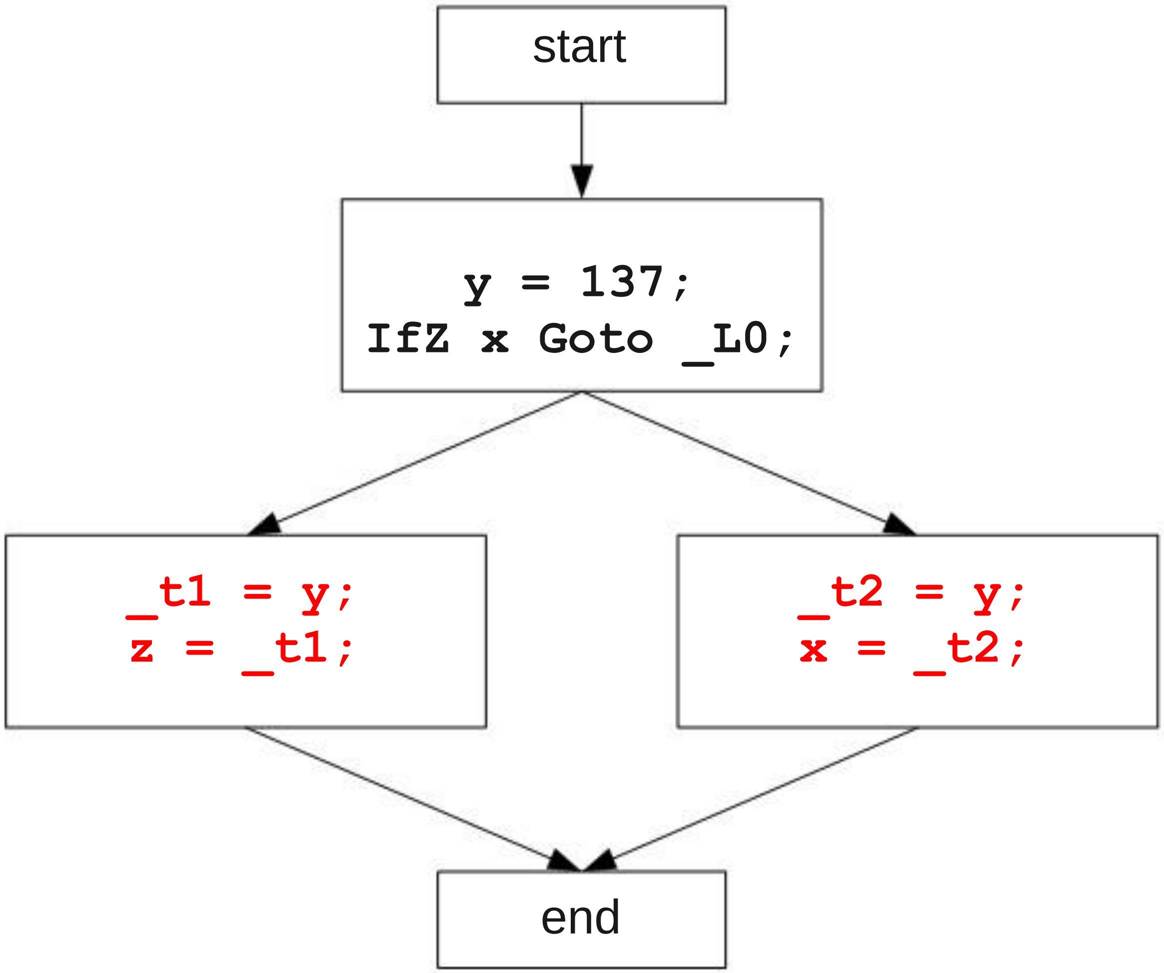
Local Optimizations

```
int main() {
    int x;
    int y;
    int z;

    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}
```



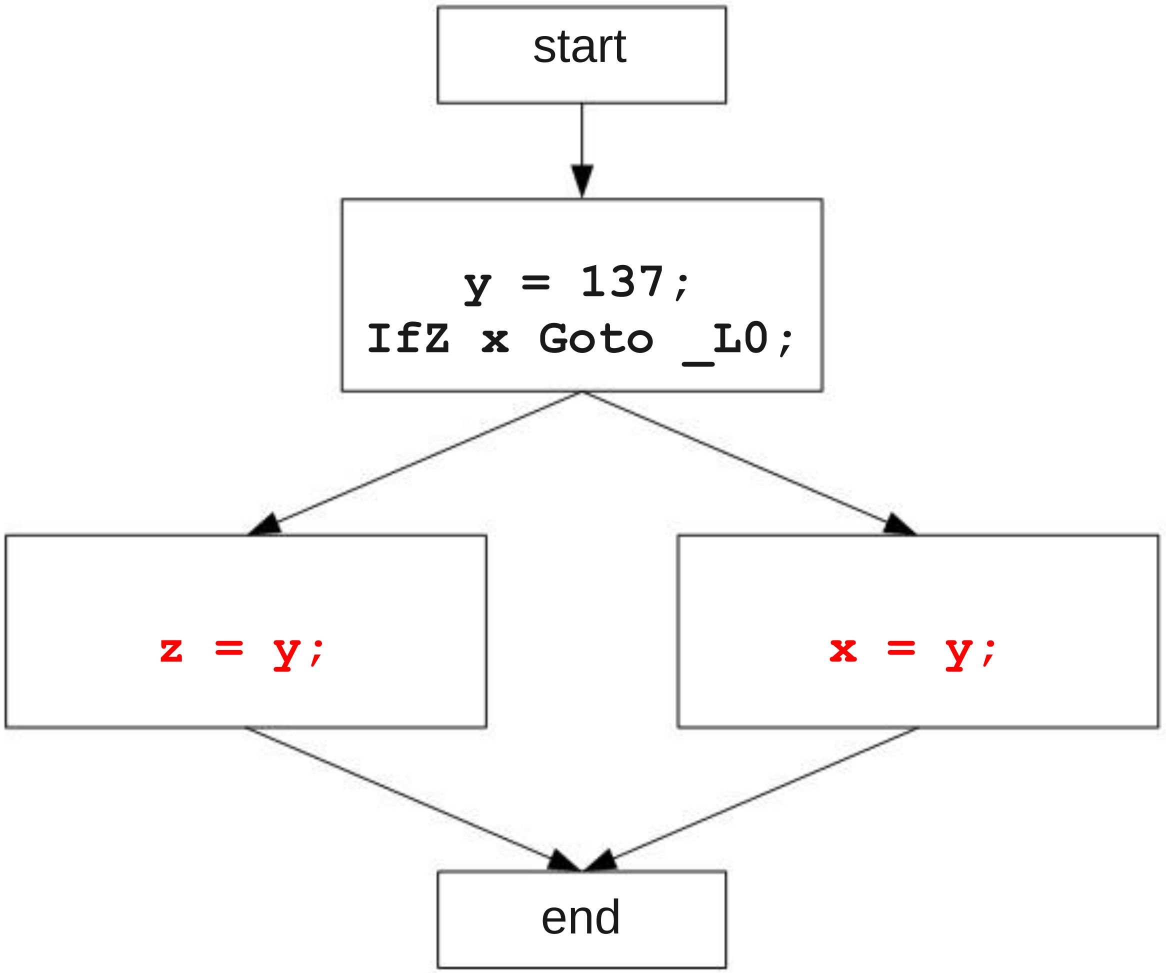
Local Optimizations

```
int main() {
    int x;
    int y;
    int z;

    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}
```



Local Optimizations

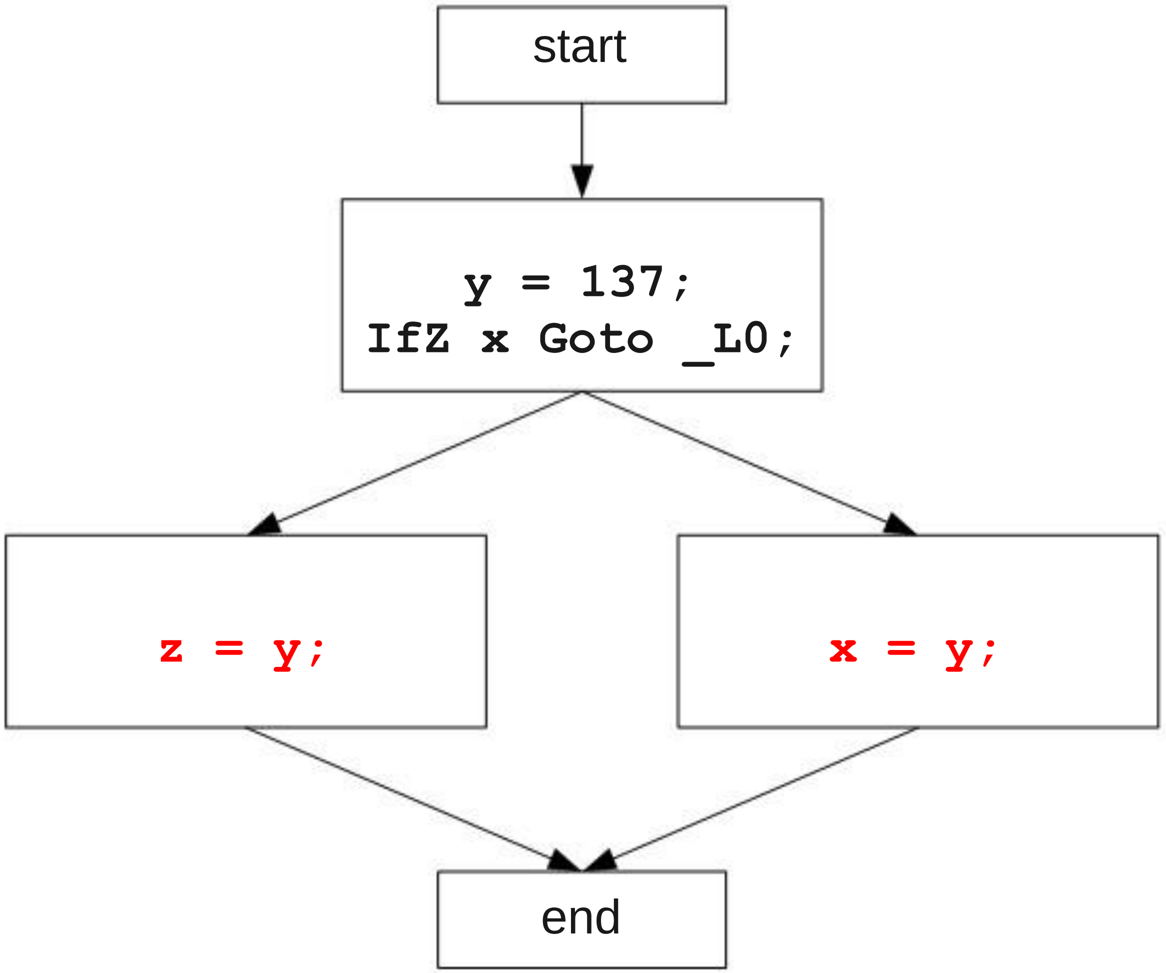
```
int main() {
    int x;
    int y;
    int z;

    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}
```



Global Optimizations

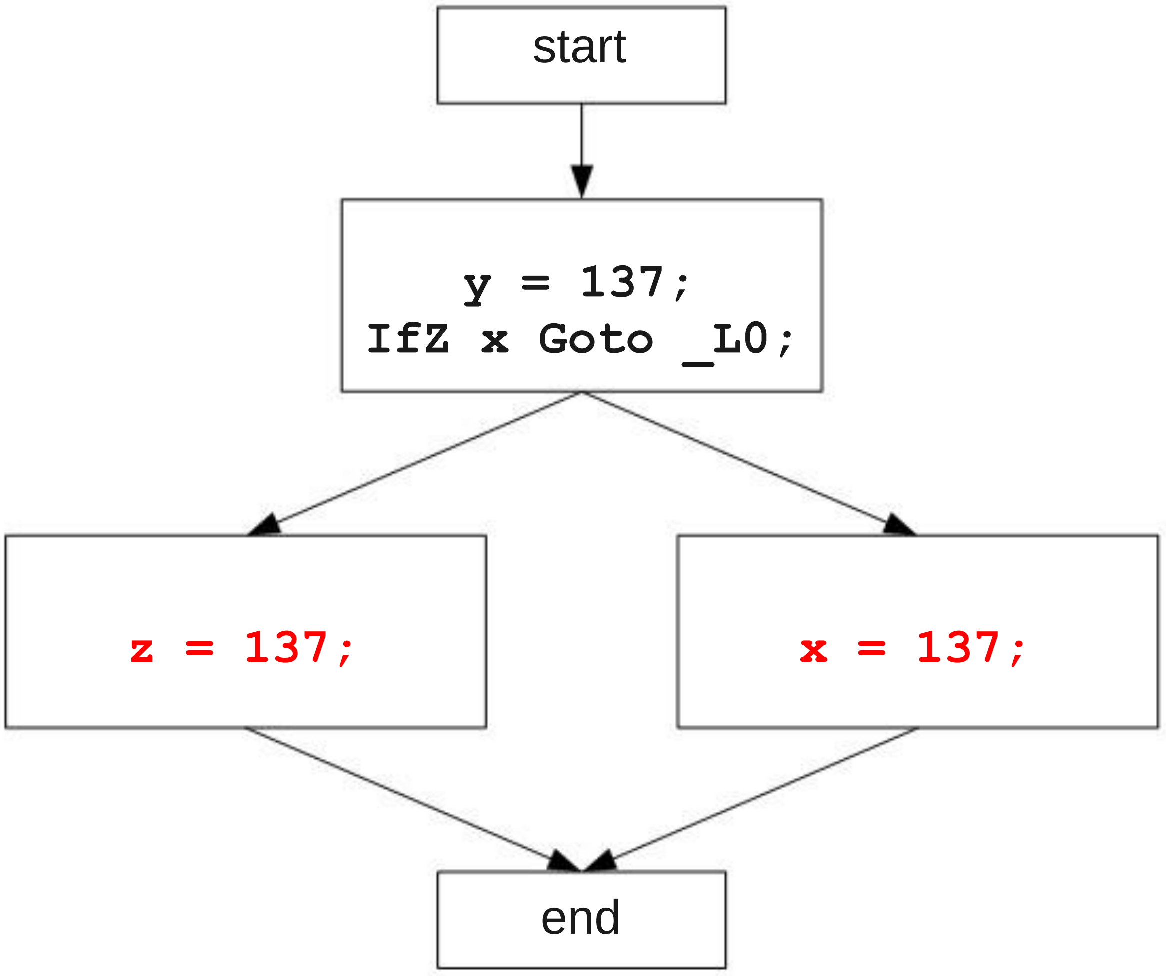
Example

```
int main() {
    int x;
    int y;
    int z;

    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}
```



Global Optimizations

```
int main() {
    int x;
    int y;
    int z;

    y = 137;
    if (x == 0)
        z = y;
    else
        x = y;
}
```



Local Optimizations

Common Subexpression Elimination

- If we have two variable assignments

```
v1 = a op b
  ...
v2 = a op b
```

and the values of **v_1**, **a**, and **b** have not changed between the assignments, rewrite the code as

```
v1 = a op b
  ...
v2 = v1
```

- Eliminates useless recalculation.

Common Subexpression Elimination

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = 4 ;
a = _tmp3 ;
_tmp4 = a + b ;
c = _tmp4 ;
_tmp5 = a + b ;
_tmp6 = *(x) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam x ;
ACall _tmp7 ;
PopParams 8 ;
```

Code Explanation

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = 4 ;
a = _tmp3 ;
_tmp4 = a + b ;
c = _tmp4 ;
_tmp5 = a + b ;
_tmp6 = *(x) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam x ;
ACall _tmp7 ;
PopParams 8 ;
```

Code Explanation

_tmp1

Allocated space

_tmp1 stores allocated memory address

4 bytes memory address

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

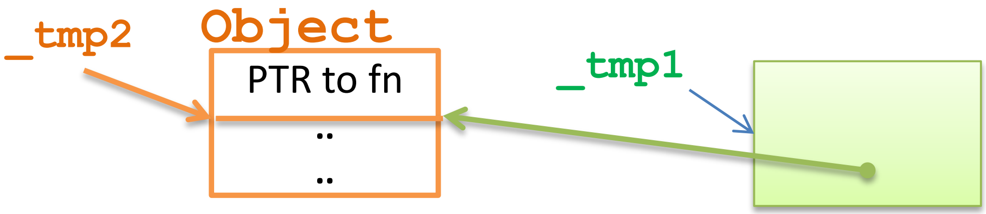
```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = 4 ;
a = _tmp3 ;
_tmp4 = a + b ;
c = _tmp4 ;
_tmp5 = a + b ;
_tmp6 = *(x) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam x ;
ACall _tmp7 ;
PopParams 8 ;
```

Code Explanation


```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = 4 ;
a = _tmp3 ;
_tmp4 = a + b ;
c = _tmp4 ;
_tmp5 = a + b ;
_tmp6 = *(x) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam x ;
ACall _tmp7 ;
PopParams 8 ;
```

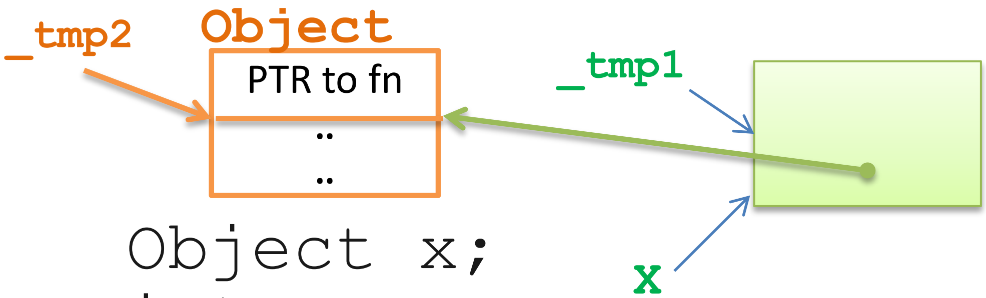
Code Explanation




```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = 4 ;
a = _tmp3 ;
_tmp4 = a + b ;
c = _tmp4 ;
_tmp5 = a + b ;
_tmp6 = *(x) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam x ;
ACall _tmp7 ;
PopParams 8 ;
```

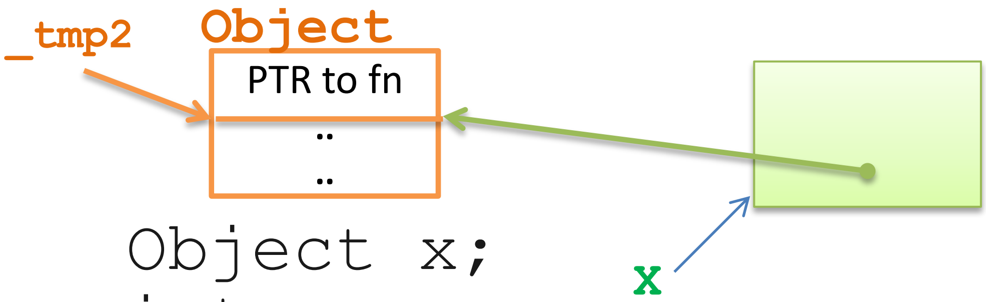
Code Explanation




```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = 4 ;
a = _tmp3 ;
_tmp4 = a + b ;
c = _tmp4 ;
_tmp5 = a + b ;
_tmp6 = *(x) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam x ;
ACall _tmp7 ;
PopParams 8 ;
```

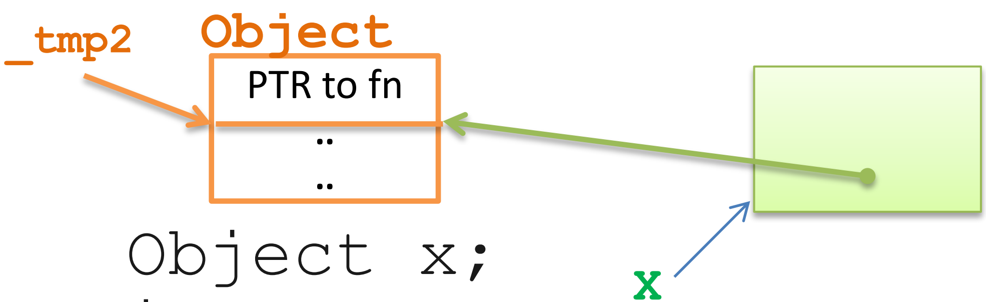
Code Explanation



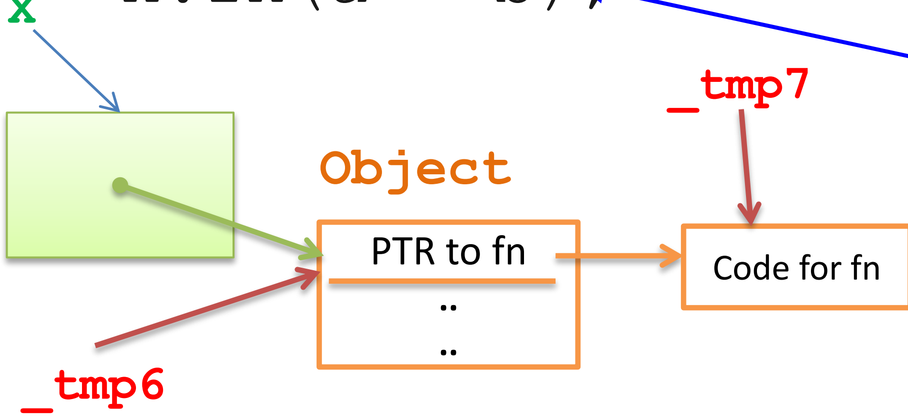

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = 4 ;
a = _tmp3 ;
_tmp4 = a + b ;
c = _tmp4 ;
_tmp5 = a + b ;
_tmp6 = *(x) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam x ;
ACall _tmp7 ;
PopParams 8 ;
```

Code Explanation




```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```




```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = 4 ;
a = _tmp3 ;
_tmp4 = a + b ;
c = _tmp4 ;
_tmp5 = a + b ;
_tmp6 = *(x) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam x ;
ACall _tmp7 ;
PopParams 8 ;
```

Follow pointer to fn

Common Subexpression Elimination

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = 4 ;
a = _tmp3 ;
_tmp4 = a + b ;
c = _tmp4 ;
_tmp5 = a + b ;
_tmp6 = *(x) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam x ;
ACall _tmp7 ;
PopParams 8 ;
```

Common Subexpression Elimination

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = 4 ;
a = _tmp3 ;
_tmp4 = a + b ;
c = _tmp4 ;
_tmp5 = _tmp4 ;
_tmp6 = *(x) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam x ;
ACall _tmp7 ;
PopParams 8 ;
```

Common Subexpression Elimination

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = 4 ;
a = _tmp3 ;
_tmp4 = a + b ;
c = _tmp4 ;
_tmp5 = _tmp4 ;
_tmp6 = *(x) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam x ;
ACall _tmp7 ;
PopParams 8 ;
```

Common Subexpression Elimination

```
Object x;
int a;
int b;
int c;

x = new Object;
a = 4;
c = a + b;
x.fn(a + b);
```

```
_tmp0 = 4 ;
PushParam _tmp0 ;
_tmp1 = LCall _Alloc ;
PopParams 4 ;
_tmp2 = Object ;
*(_tmp1) = _tmp2 ;
x = _tmp1 ;
_tmp3 = _tmp0 ;
a = _tmp3 ;
_tmp4 = a + b ;
c = _tmp4 ;
_tmp5 = _tmp4 ;
_tmp6 = *(x) ;
_tmp7 = *(_tmp6) ;
PushParam _tmp5 ;
PushParam x ;
ACall _tmp7 ;
PopParams 8 ;
```

Next class: More optimization on this code