50.051 Programming Language Concepts

W12-S3 Intermediate Code Generation

Matthieu De Mari

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

Code is legal!

Definition (Legal Code):

A **legal code** is a program that adheres to all the rules of a programming language. It passed through various stages, including:

- **Lexical Analysis**
- **Syntax Analysis**
- **Semantic Analysis**

Property: A code is considered legal if it satisfies the requirements of lexical analysis, syntax analysis, semantic analysis.

A code that is deemed legal should be able to run correctly, after it has been transformed into machine code.

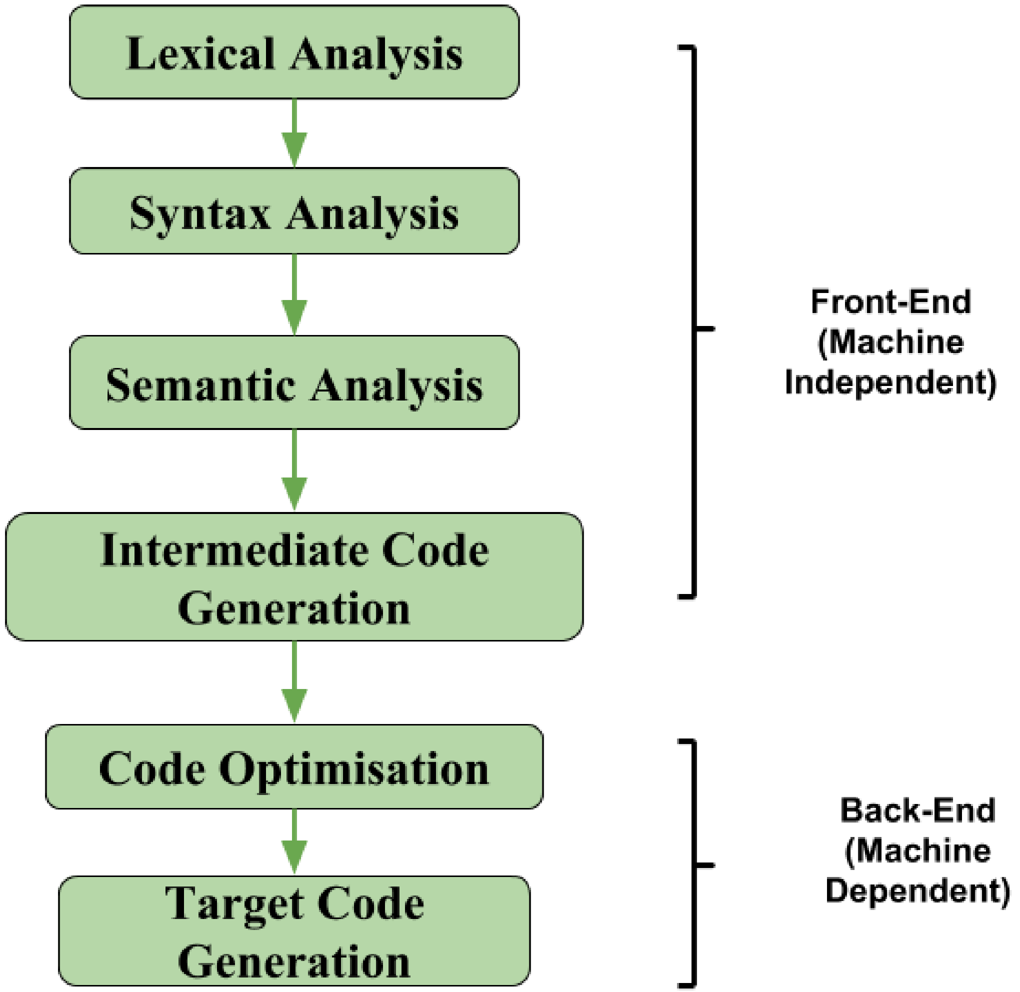
The middle-end of a compiler

Definition (The middle-end part of a compiler):

The **middle-end of a compiler** follows the front-end analysis and it consists of a series of operations and transformations to optimize and improve its efficiency.

It involves tasks, such as:

- **Intermediate code generation**
- **Code optimization**,
- and **Data-flow analysis**.

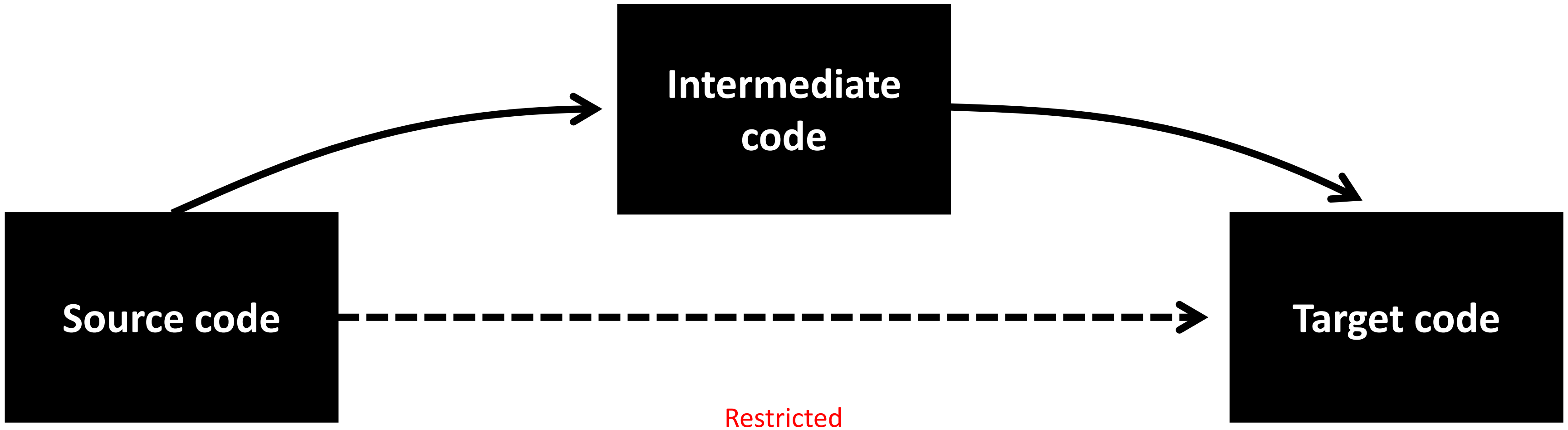


Intermediate code generation

Definition (Intermediate code generation):

Intermediate code generation is the process of transforming the source code into a code that is more abstract and closer to machine language.

Making a direct jump from source code to target code might prove difficult.



Intermediate code generation

Definition (Intermediate code generation):

Intermediate code generation is the process of transforming the source code into a code that is more abstract and closer to machine language.

Making a direct jump from source code to target code might prove difficult.

For this reason, an intermediate code representation is often easier to manipulate and allows for optimization.

The intermediate code is not specific to any particular hardware or operating system and can be easily transformed into the final machine code (**Intermediate code is roughly a high-level assembly code?**).

Three-address code representation

Definition (Three-address code representation):

Three-address code (or TAC) is a low-level intermediate code representation used by compilers to facilitate optimization and code generation.

It is called **"three-address"** because each instruction in the code **can have at most three operands**.

A typical three-address code instruction has the following format:

$$operand1 = operand2 \ operator \ operand3$$

Three-address code representation

For instance, the C code below can be transformed…

```c
#include <stdio.h>

int main() {
   int x = 10;
   int y = x + 5;
   printf("The value of y is %d\n", y);
   return 0;
}
```

…into its equivalent three-address code representation.

```
t1 = 10
t2 = t1 + 5
y = t2
printf("The value of y is %d\n", y)
```

Overview

Our plan for today!

- Introduction to TAC.
- TAC for simple expressions.
- TAC for functions and function calls.
- TAC for objects.
- TAC for arrays.
- Generating TAC.
- A few low-level details.

A quick note before we start

An important note before we start

- When generating IR at this level, you do not need to worry about optimizing it.
- It is okay to generate IR that has lots of unnecessary assignments, redundant computations, uses many temporary variables, etc.

- In the next lecture, we will discuss how to optimize IR code.
- Optimization is tricky, but interesting!

CFG for TAC

- Technically, the three-address code has a syntax and CFG of its own.

P → SP | ϵ

S → id := id op id

S → id := op id

S → id := id

S → push id

S → if id goto L

S → L:

S → jump L

id → (any identifier name)

id → (any constant literal)

op → (any basic operator
e.g. +, -, *, /, ==, <,
||, &&, etc.)

L → L0 | L1 | … (block name)

Normally, many more production rules for S…

Basic block in TAC

Definition (basic block in TAC):

A **basic block** consists of a sequence of instructions in TAC, with:

- No labels (except at the first instruction),
- No jumps (except at the last instruction of the block).

Core idea behind basic blocks:

- Cannot jump in the middle of a basic block, only the beginning,
- Cannot jump out of a block, except at the end of it,
- Basic block is then a single-entry and single-exit code segment.

Basic block in TAC

A basic block always writes as:

- Start with a **block name**, by convention, in the form of Ln, with n an integer.
- Has **elementary operations** in TAC format.
- Could have **conditional structures**, using if.
- **Goto/jump** could reference current block or another block.

For instance

```
L1:
    w := x - 1
    z := w > 0
    if z goto L1
    jump L2
```

Basic block in TAC

Question: Which C code could have potentially generated this TAC?

For instance

```
L1:
    w := x - 1
    z := w > 0
    if z goto L1
    jump L2
```

Basic block in TAC

Question: Which C code could have potentially generated this TAC?

```
int x = 4;
…                                   (Some code L0)
do {
        x = x - 1
} while (x > 0)
…                   (Some more code in L2)
```

For instance

L1:

w := x - 1

z := w > 0

if z goto L1

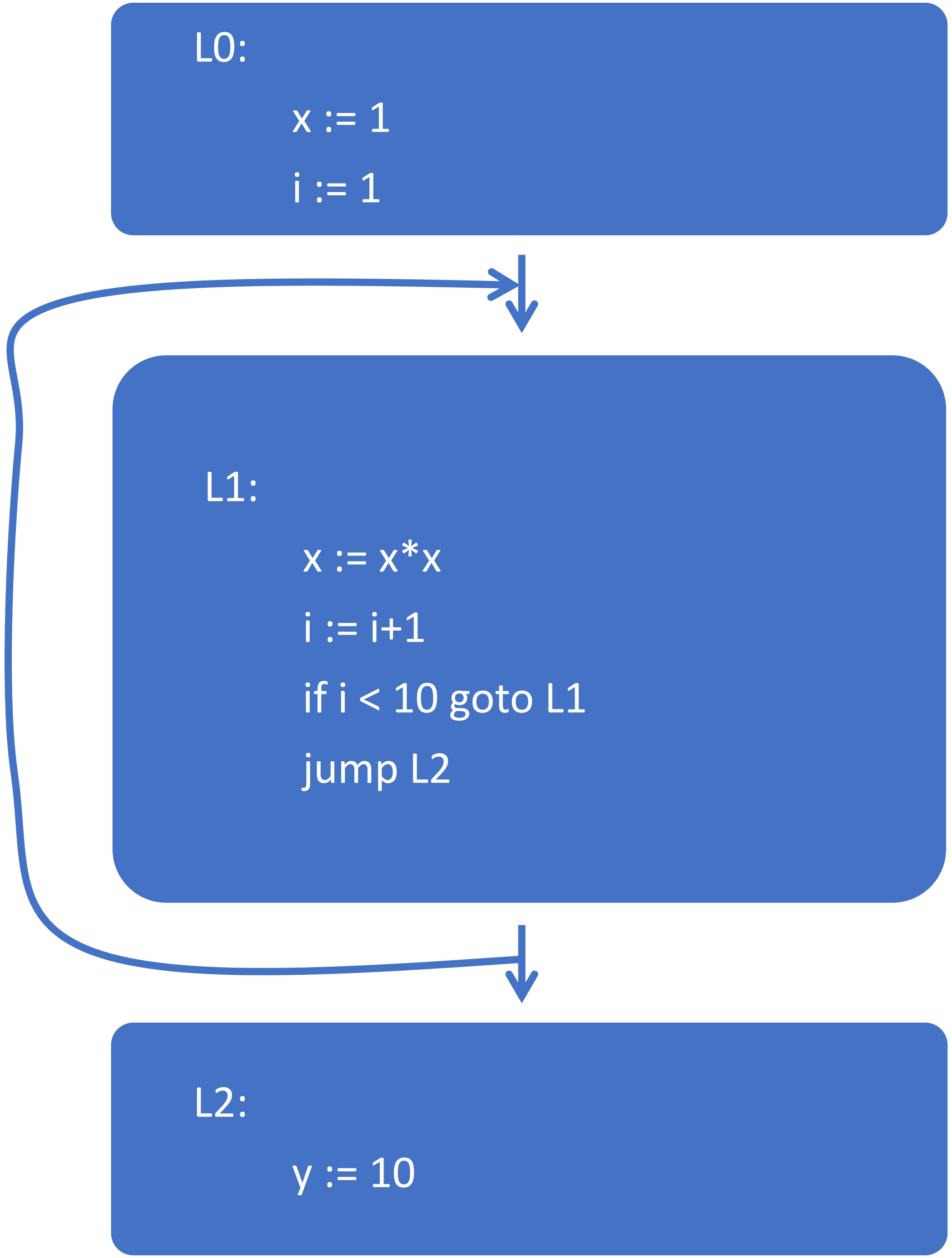
jump L2

Control flow graph

Definition (control-flow graph):

A **control-flow graph** is a directed graph with basic blocks as nodes, and an edge from block A to block B if the execution can pass from the last instruction in A to the first instruction in B.

- E.g., the last instruction in A is jump LB
- E.g., execution can fall-through from block A to block B



TAC for simple expressions

C code

```
int a;
int b;
a = 3
b = 1
c = 5 + 2*b
d = a*a + b*c
```

TAC code

```
_t0 := 3
_t1 := 1
_t2 := 2 * _t1
_t3 := 5 + _t2
_t4 = _t0 * _t0
_t5 = _t1 * _t3
_t6 = _t4 + _t5
```

TAC for simple expressions

Observations

- The “three” in “three-address code” refers to the upmost number of operands in any instruction.
- Could technically have instructions with only two operands.
- Evaluating an expression with more than three subexpressions requires the introduction of temporary variables.

Question: who holds the key to how to deconstruct a long mathematical expression into smaller three-operands expressions with the right precedence order?

TAC for simple expressions

Observations

- The “three” in “three-address code” refers to the upmost number of operands in any instruction.
- Could technically have instructions with only two operands.
- Evaluating an expression with more than three subexpressions requires the introduction of temporary variables.

Question: who holds the key to how to deconstruct a long mathematical expression into smaller three-operands expressions with the right precedence order?

Answer: Your parse tree/abstract syntax tree from earlier!

TAC for simple expressions

C code

int a;

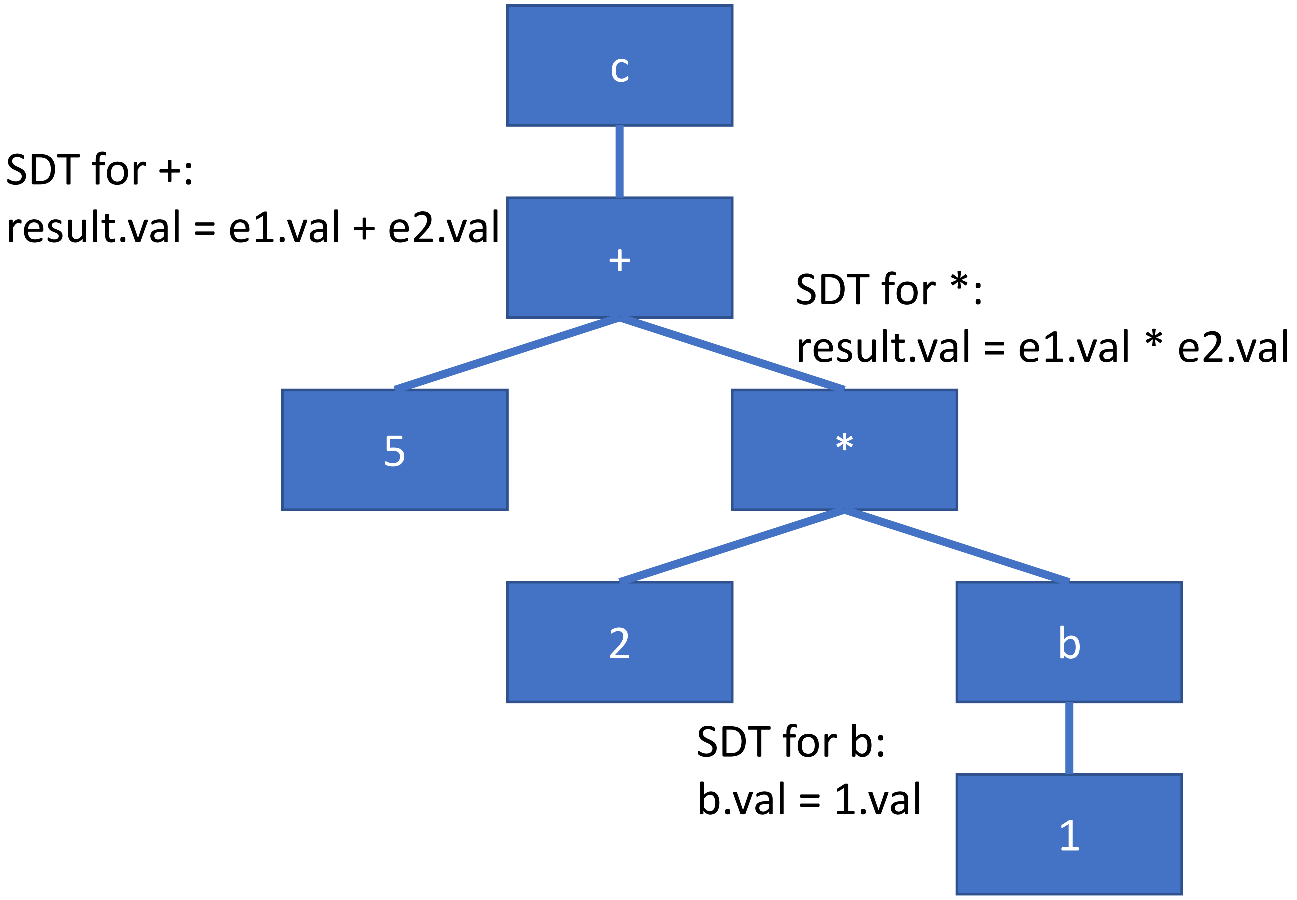
int b;

a = 3

b = 1

*c = 5 + 2*b*

*d = a*a + b*c*

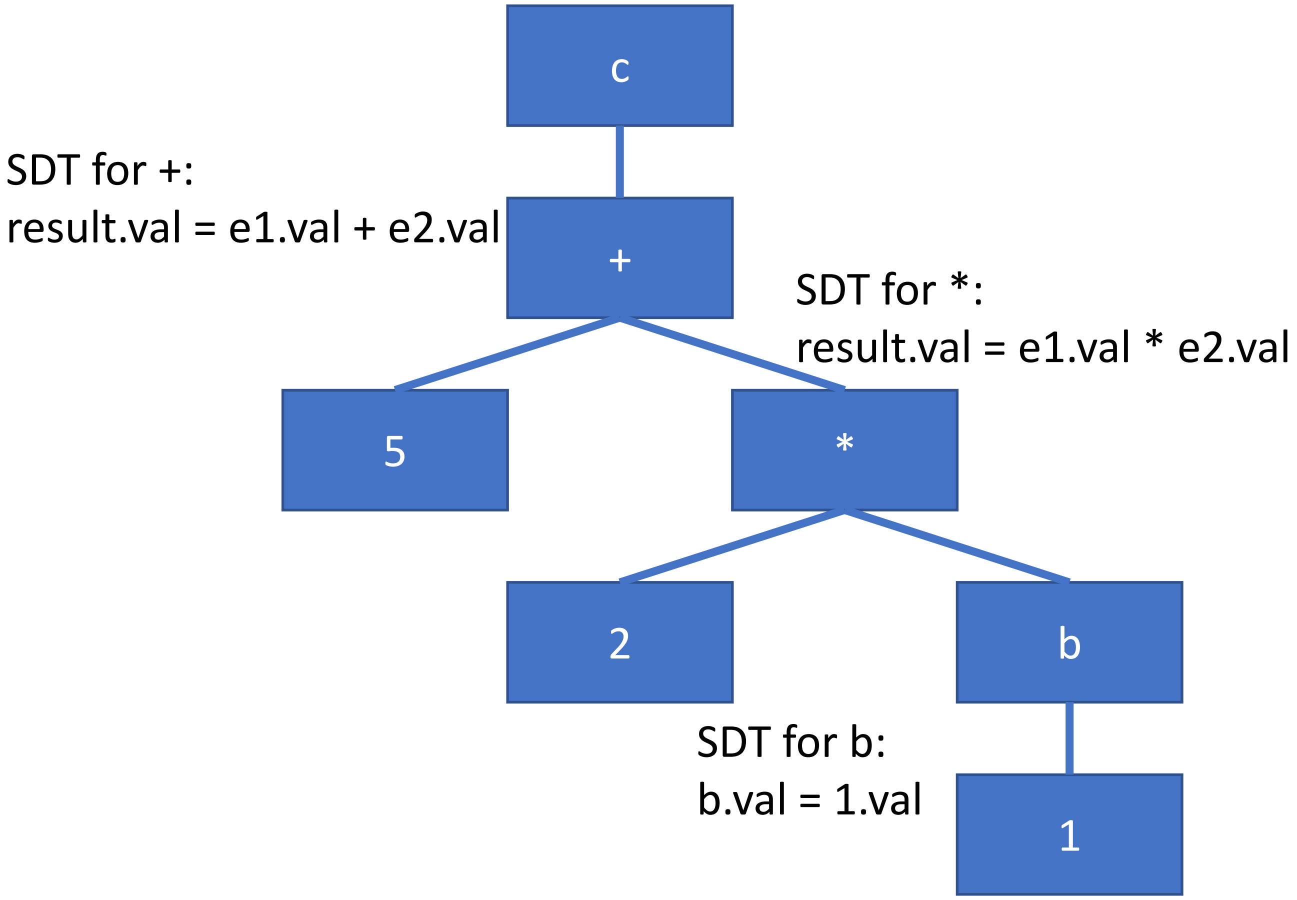


TAC for simple expressions

Answer: Your abstract syntax tree from earlier!

Remember, we have an AST,

- Annotated with elementary operations (syntax analysis),
- Annotated with scope information (semantic analysis),
- And annotated with type information (semantic analysis).



Restricted

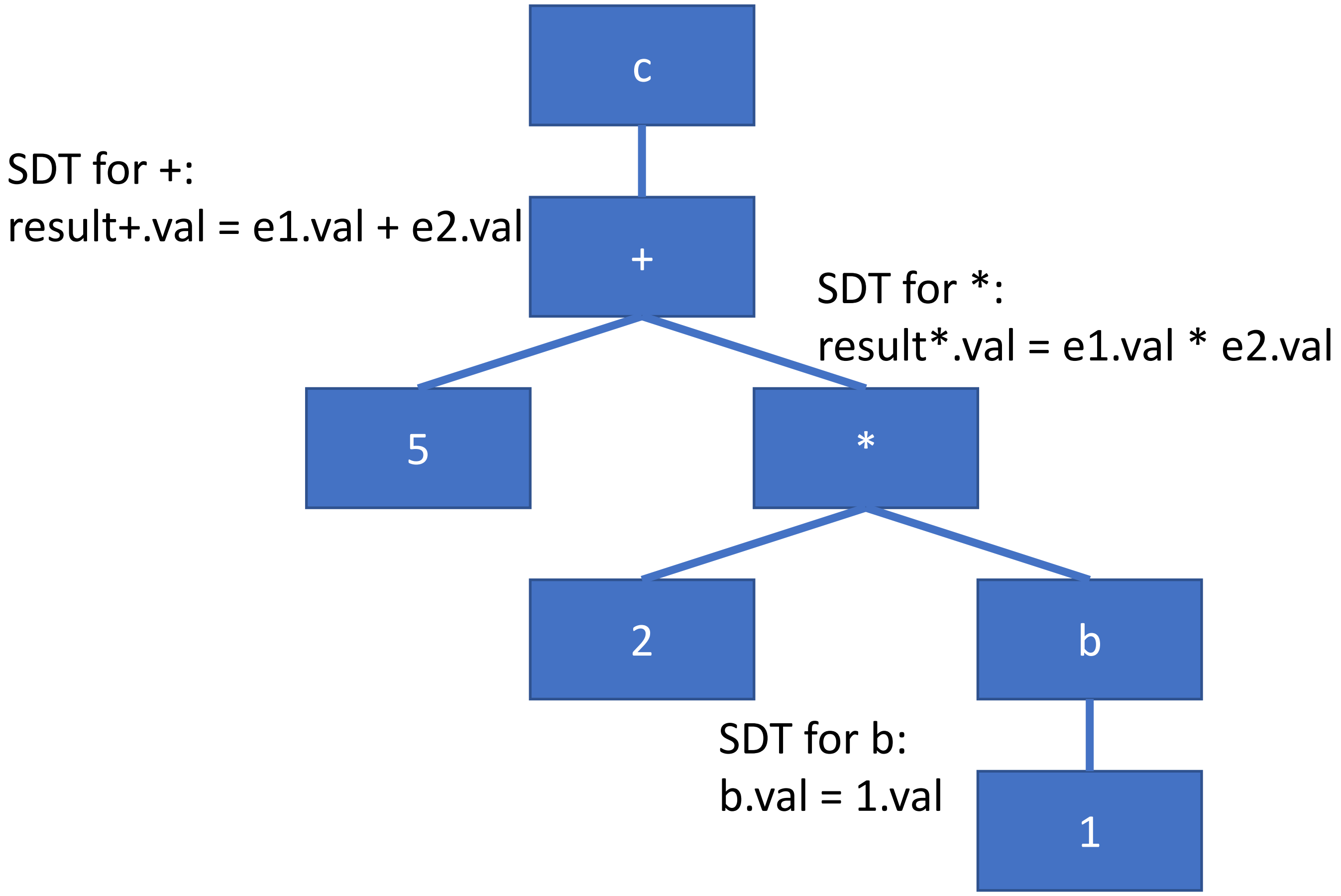
TAC for simple expressions

Collecting each of the SDT instructions in the AST using our recursive descent tree algorithm/DFS gives…

b.val = 1.val

*result.val = e1.val * e2.val*

result.val = e1.val + e2.val



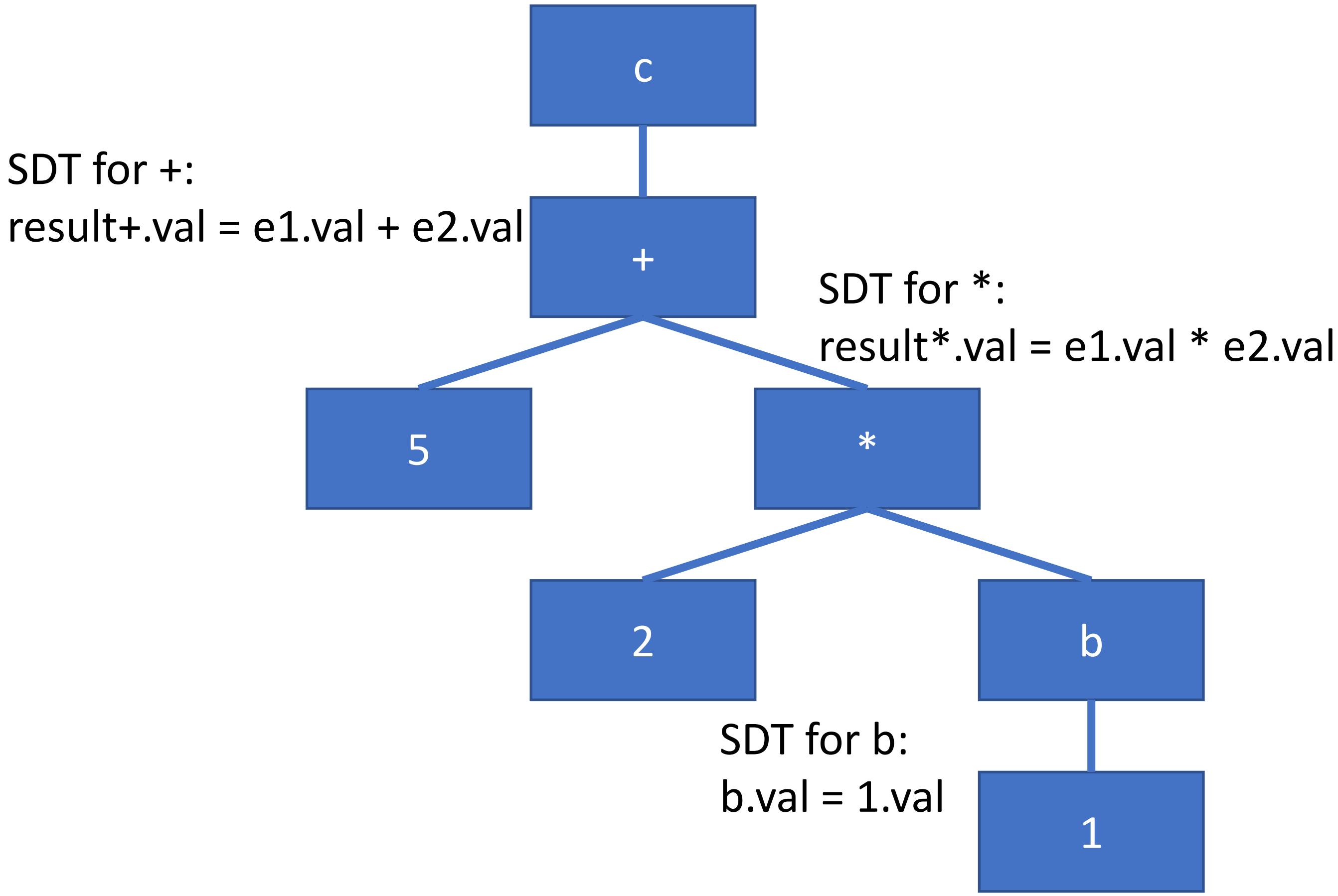
TAC for simple expressions

Using _tx notations in order to disambiguate and recognizing connections…

_t0.val = 1.val

*_t1.val = 2.val * _t0.val*

_t2.val = 5.val + _t1.val



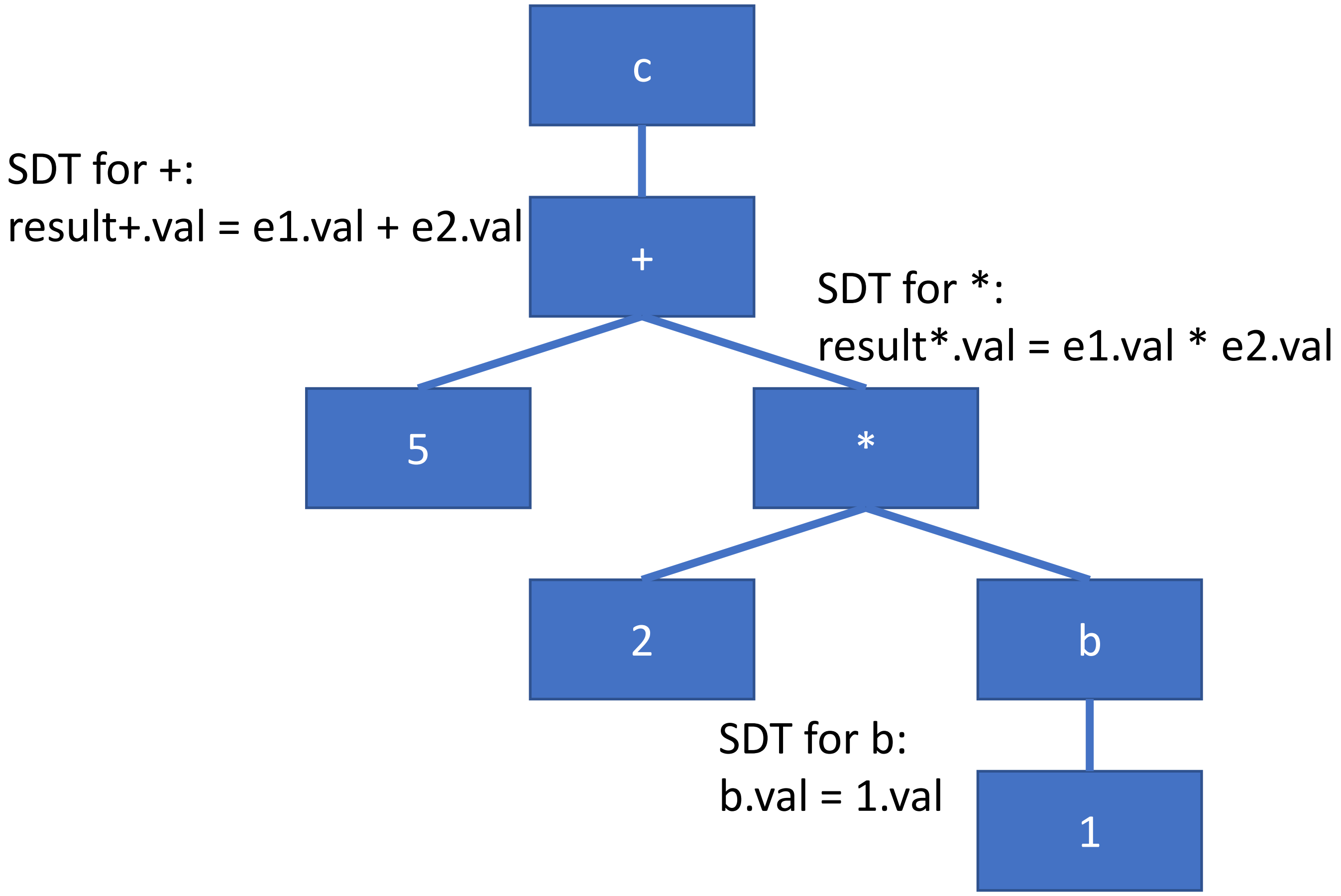
TAC for simple expressions

Finally, recognizing that constant.val = constant (e.g. 5.val = 5)

_t0.val = 1

*_t1.val = 2 * _t0.val*

_t2.val = 5 + _t1.val



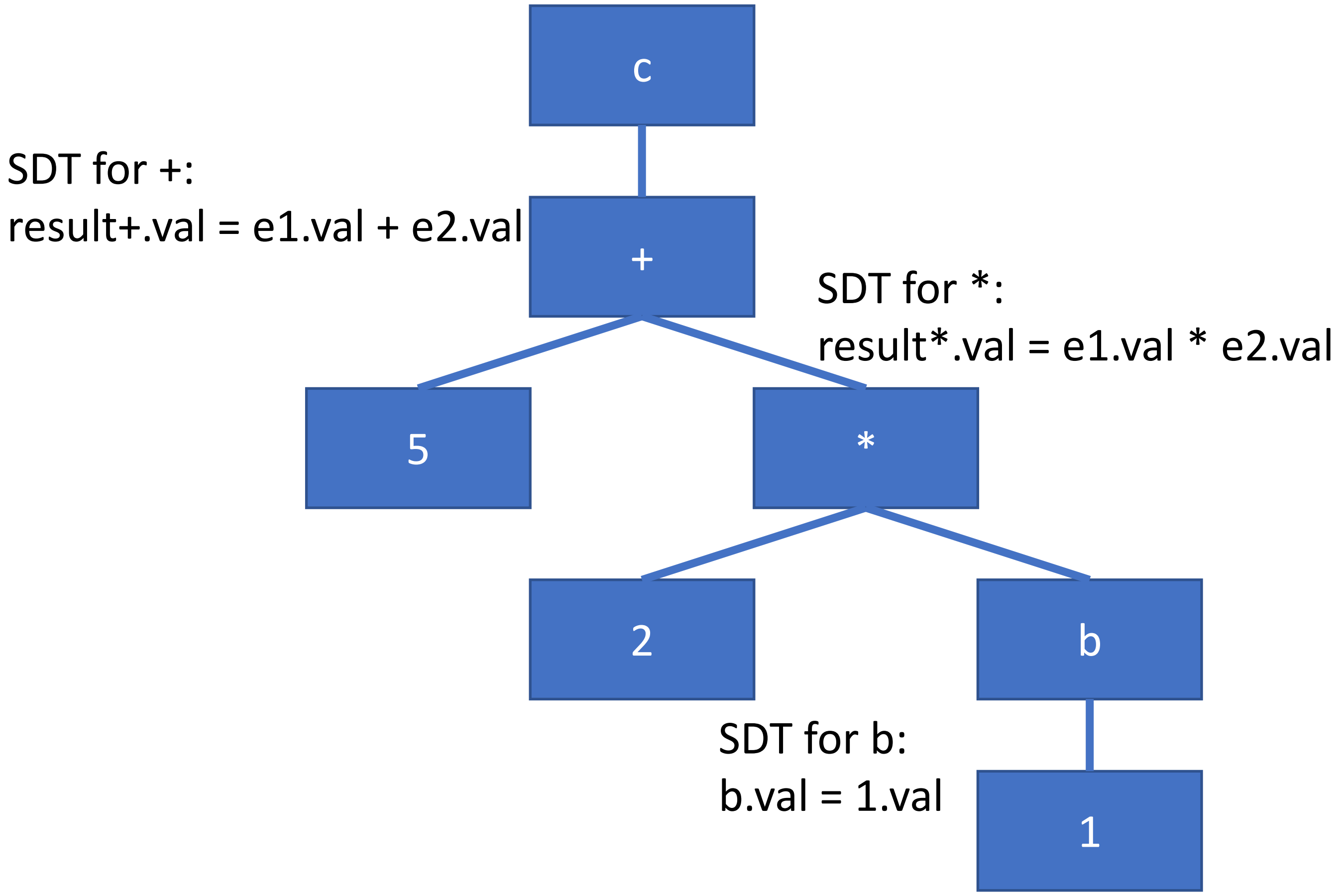
TAC for simple expressions

Repeat for the AST computing the value of d and chain instructions together!

_t0.val = 1

*_t1.val = 2 * _t0.val*

_t2.val = 5 + _t1.val



TAC generation for SDT expressions

Our SDT expressions are quite close to the TAC expressions already.

All we need is

- A mapping to translate our SDT elementary operations into their TAC equivalents (SDT: e.val = a.val + b.val → TAC: e:= a + b).
- A function that will correctly link variables using the edges of our abstract syntax tree (easily done).
- A function that will open registers _*tn* and keep a counter on n.

Could be easily implemented (but would require an implementation of an abstract syntax tree object during parsing, which we did not do!).

TAC generation for SDT expressions

Core idea for implementation: define a code generation, or *cgen()* function, whose behaviour is described below in a few scenarios.

cgen(k) = { // here k refers to a constant literal

Choose a new temporary _tn using our counter n;

Increment n by 1;

Emit (_tn = k);

Return _tn;

}

TAC generation for SDT expressions

Core idea for implementation: define a code generation, or *cgen()* function, whose behaviour is described below in a few scenarios.

cgen(id) = { // here id refers to an identifier (could be a _tn)

Choose a new temporary _tn using our counter n;

Increment n by 1;

Emit (_tn = id);

Return _tn;

}

TAC generation for SDT expressions

Core idea for implementation: define a code generation, or *cgen()* function, whose behaviour is described below in a few scenarios.

```
cgen(id = k) = { // here id refers to an identifier (could be a _tn)
// and k refers to a constant literal.
Emit (id = k);
Return null;
}
```

TAC generation for SDT expressions

Core idea for implementation: define a code generation, or *cgen()* function, whose behaviour is described below in a few scenarios.

cgen(e1 + e2) = {

Choose three new temporary t using our counter n
(we shall name them as _tn, _t(n+1), _t(n+2));

Increment n by 3;

Let _tn = cgen(e1);

Let _t(n+1) = cgen(e2);

Emit (_t(n+2) = _tn + _t(n+1));

Return _t(n+2);

}

TAC generation for SDT expressions

For instance, let us assume we have the expression 5 + x and n = 0.

cgen(e1 + e2) = {

Choose three new temporary t using our counter n
(we shall name them as _tn, _t(n+1), _t(n+2));

Increment n by 3;

Let _tn = cgen(e1);

Let _t(n+1) = cgen(e2);

Emit (_t(n+2) = _tn + _t(n+1));

Return _t(n+2);

}

TAC generation for SDT expressions

For instance, let us assume we have the expression 5 + x and n = 0.

cgen(5 + x) = {

Choose three new temporary t using our counter n
(we shall name them as _tn, _t(n+1), _t(n+2));

Increment n by 3;

Let _tn = cgen(5);

Let _t(n+1) = cgen(x);

Emit (_t(n+2) = _tn + _t(n+1));

Return _t(n+2);

}

TAC generation for SDT expressions

For instance, let us assume we have the expression 5 + x and n = 0.

cgen(5 + x) = {

Choose three new temporary t using our counter n
(we shall name them as _t0, _t1, _t2);

Increment n by 3;

Let _t0 = cgen(5);

Let _t1 = cgen(x);

Emit (_t2 = _t0 + _t1);

Return _t2;

}

TAC generation for SDT expressions

For instance, let us assume we have the expression 5 + x and n = 3.

cgen(5 + x) = {

Choose three new temporary t using our counter n
(we shall name them as _t0, _t1, _t2);

Increment n by 3;

Let _t0 = cgen(5);

Let _t1 = cgen(x);

Emit (_t2 = _t0 + _t1);

Return _t2;

}

TAC generation for SDT expressions

For instance, let us assume we have the expression 5 + x and n = 3.

cgen(5 + x) = {

Choose three new temporary t using our counter n
(we shall name them as _t0, _t1, _t2);

Increment n by 3;

Let _t0 = cgen(5);

Let _t1 = cgen(x);

Emit (_t2 = _t0 + _t1);

Return _t2;

}

cgen(5) = {
Choose a new temporary _tn using our counter n;
Increment n by 1;
Emit (_tn = k);
Return _tn;
}

TAC generation for SDT expressions

For instance, let us assume we have the expression 5 + x and n = 4.

cgen(5 + x) = {

Choose three new temporary t using our counter n
(we shall name them as _t0, _t1, _t2);

Increment n by 3;

Let _t0 = cgen(5);

Let _t1 = cgen(x);

Emit (_t2 = _t0 + _t1);

Return _t2;

}

cgen(5) = {
Choose _t4
Increment n by 1;
Emit (_tn = k);
Return _tn;
}

TAC generation for SDT expressions

For instance, let us assume we have the expression 5 + x and n = 4.

cgen(5 + x) = {

Choose three new temporary t using our counter n
(we shall name them as _t0, _t1, _t2);

Increment n by 3;

Let _t0 = cgen(5);

Let _t1 = cgen(x);

Emit (_t2 = _t0 + _t1);

Return _t2;

}

cgen(5) = {
Choose _t4
Increment n by 1;
Emit (_t4 = 5);
Return _t4;
}

TAC code
_t4 := 5

TAC generation for SDT expressions

For instance, let us assume we have the expression 5 + x and n = 4.

cgen(5 + x) = {

Choose three new temporary t using our counter n
(we shall name them as _t0, _t1, _t2);

Increment n by 3;

Let _t0 = cgen(5);

Let _t1 = cgen(x);

Emit (_t2 = _t0 + _t1);

Return _t2;

}

cgen(5) = {
Choose _t4
Increment n by 1;
Emit (_t4 = 5);
Return _t4;
}

TAC code
_t4 := 5
_t0 := _t4

Restricted

TAC generation for SDT expressions

For instance, let us assume we have the expression 5 + x and n = 3.

cgen(5 + x) = {

Choose three new temporary t using our counter n
(we shall name them as _t0, _t1, _t2);

Increment n by 3;

Let _t0 = cgen(5);

Let _t1 = cgen(x);

Emit (_t2 = _t0 + _t1);

Return _t2;

}

cgen(id) = {
Choose a new temporary _tn using our counter n;
Increment n by 1;
Emit (_tn = id);
Return _tn;
}

TAC generation for SDT expressions

For instance, let us assume we have the expression 5 + x and n = 5.

cgen(5 + x) = {

Choose three new temporary t using our counter n (we shall name them as _t0, _t1, _t2);

Increment n by 3;

Let _t0 = cgen(5);

Let _t1 = cgen(x);

Emit (_t2 = _t0 + _t1);

Return _t2;

}

cgen(x) = {

Choose _t5

Increment n by 1;

Emit (_tn = id);

Return _tn;

}

TAC code

_t4 := 5

_t0 := _t4

TAC generation for SDT expressions

For instance, let us assume we have the expression 5 + x and n = 4.

cgen(5 + x) = {

Choose three new temporary t using our counter n (we shall name them as _t0, _t1, _t2);

Increment n by 3;

Let _t0 = cgen(5);

Let _t1 = cgen(x);

Emit (_t2 = _t0 + _t1);

Return _t2;

}

cgen(x) = {

Choose _t5

Increment n by 1;

Emit (_t5 = x);

Return _tn;

}

TAC code

```
_t4 := 5
_t0 := _t4
_t5 := x
```

TAC generation for SDT expressions

For instance, let us assume we have the expression 5 + x and n = 4.

cgen(5 + x) = {

Choose three new temporary t using our counter n
(we shall name them as _t0, _t1, _t2);

Increment n by 3;

Let _t0 = cgen(5);

Let _t1 = cgen(x);

Emit (_t2 = _t0 + _t1);

Return _t2;

}

cgen(x) = {

Choose _t5

Increment n by 1;

Emit (_t5 = x);

Return _t5;

}

TAC code

_t4 := 5
_t0 := _t4
_t5 := x
_t1 := _t5

TAC generation for SDT expressions

For instance, let us assume we have the expression 5 + x and n = 4.

cgen(5 + x) = {

Choose three new temporary t using our counter n (we shall name them as _t0, _t1, _t2);

Increment n by 3;

Let _t0 = cgen(5);

Let _t1 = cgen(x);

Emit (_t2 = _t0 + _t1);

Return _t2;

}

TAC code

```
_t4 := 5
_t0 := _t4
_t5 := x
_t1 := _t5
_t2 := _t0 + _t1
```

TAC generation for SDT expressions

For instance, let us assume we have the expression 5 + x and n = 4.

cgen(5 + x) = {

Choose three new temporary t using our counter n
(we shall name them as _t0, _t1, _t2);

Increment n by 3;

Let _t0 = cgen(5);

Let _t1 = cgen(x);

Emit (_t2 = _t0 + _t1);

Return _t2;

In case there are more operations after this!

}

TAC code

```
_t4 := 5
_t0 := _t4
_t5 := x
_t1 := _t5
_t2 := _t0 + _t1
```

Restricted

Remember what we said earlier

A quick note before we start

An important note before we start

- When generating IR at this level, you do not need to worry about optimizing it.
- It is okay to generate IR that has lots of unnecessary assignments, redundant computations, uses many temporary variables, etc.

- In the next lecture, we will discuss how to optimize IR code.
- Optimization is tricky, but interesting!

TAC code
(not optimal but works!)

```
_t4 := 5
_t0 := _t4
_t5 := x
_t1 := _t5
_t2 := _t0 + _t1
```

Restricted

Practice: TAC for booleans

Question 1: Following our idea for *cgen(e1 + e2)*, how would you define a cgen() for Boolean operations, e.g. *cgen(e1 <= e2)?*

Remember that the CFG of the TAC languages allows for ==, < and || operations, but not <=!

Question 2: Using our *cgen()* functions from earlier, how would you encode in TAC the operation “(5 + x) <= 10”? Show your steps.

Practice: TAC for booleans

Question 1: Following our idea for *cgen(e1 + e2)*, how would you define a cgen() for Boolean operations, e.g. *cgen(e1 <= e2)?*

Remember that the CFG of the TAC languages allows for ==, < and || operations, but not <=!

Question 2: Using our *cgen()* functions from earlier, how would you encode in TAC the operation “(5 + x) <= 10”? Show your steps.

Answer: to be shown on board.

TAC for control structures

Let us now assume that we have figured our TAC *cgen()* functions for all Boolean operations.

Question: How would I encode an if statement in TAC code?

if(x < y)

    z = x;

else

    z = y;

*z = z*z;*

TAC for control structures

Let us now assume that we have figured our TAC *cgen()* functions for all Boolean operations.

Question: How would I encode an if statement in TAC code?

if(x < y)

    z = x;

else

    z = y;

*z = z*z;*

What could *cgen(if S1 S2 else S3)* be, with S1, S2 and S3 being statements of some sort?

Restricted

TAC for control structures

Let us now assume that we have figured our TAC *cgen()* functions for all Boolean operations.

Question: How would I encode an if statement in TAC code?

if(x < y)

z = x;

else

z = y;

*z = z*z;*

What could *cgen(if S1 S2 else S3)* be, with S1, S2 and S3 being statements of some sort?

Remember two things:

- This if statement could be represented as a control-flow graph.

TAC for control structures

Let us now assume that we have figured our TAC *cgen()* functions for all Boolean operations.

Question: How would I encode an if statement in TAC code?

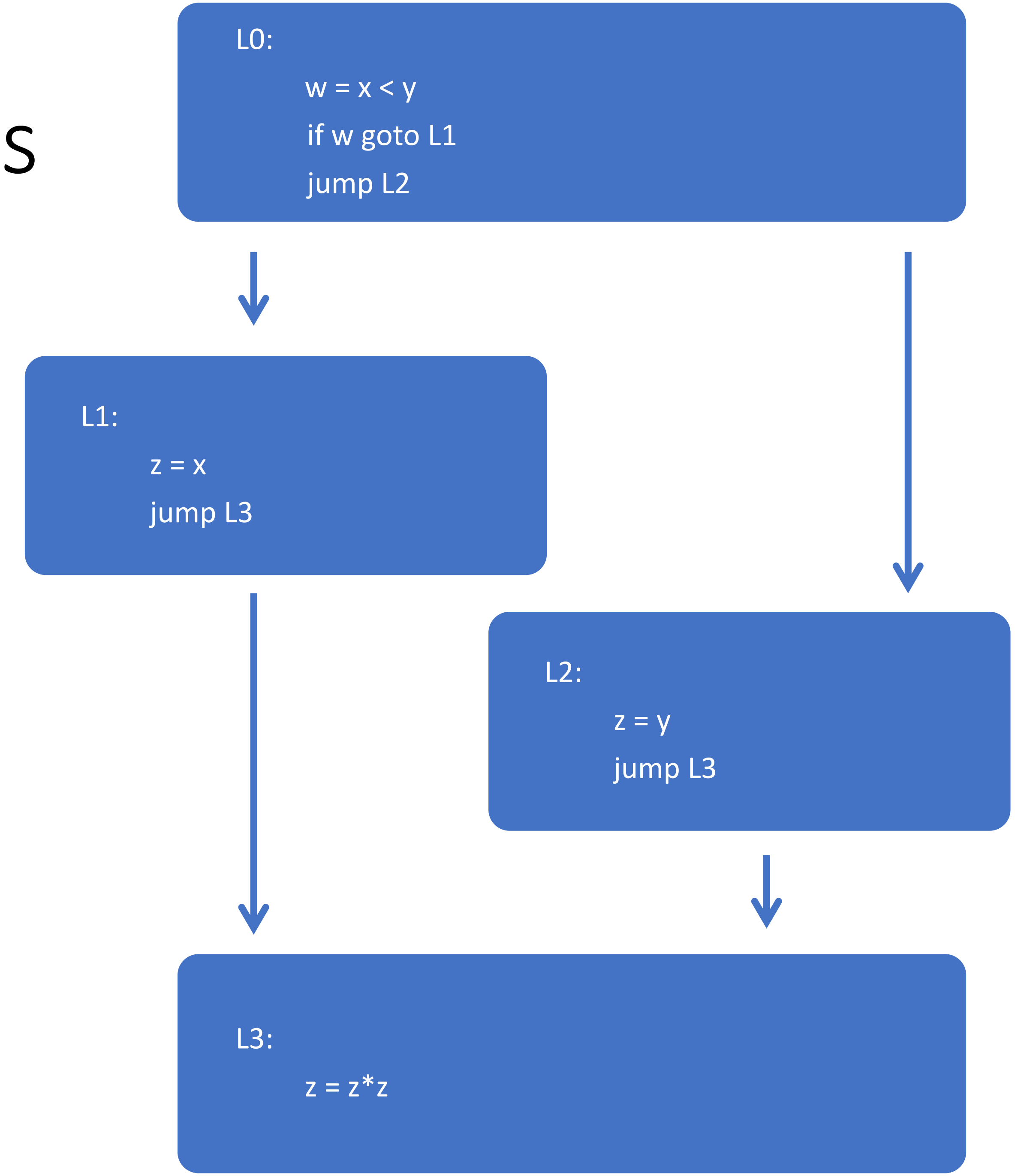
if(x < y)

    z = x;

else

    z = y;

*z = z*z;*



Restricted

TAC for control structures

Let us now assume that we have figured our TAC *cgen()* functions for all Boolean operations.

Question: How would I encode an if statement in TAC code?

if(x < y)

z = x;

else

z = y;

*z = z*z;*

What could *cgen(if S1 S2 else S3)* be, with S1, S2 and S3 being statements of some sort?

Remember two things:

- This if statement could be represented as a control-flow graph.
- Our TAC grammar allows for

$$S \rightarrow \textit{if id goto } L$$

$$S \rightarrow L:$$

$$S \rightarrow \textit{jump } L$$

TAC for control structures

cgen(if S1 S2 else S3) = {

Choose three new temporary t using our counter n. Increment n by 3;

Let _tn = cgen(S1);

Let _t(n+1) = cgen(S2); ?

Let _t(n+2) = cgen(S3); ?

…

}

TAC for control structures

We need a second counter k, which will keep track of the block indexes for defining some Lk block labels. Add it along with n.

TAC for control structures

TAC system counters: n, k.

cgen(if S1 S2 else S3) = {

Choose three new temporary t using our counter n. Increment n by 3;

Let _tn = cgen(S1);

Choose four new block labels Lk. Increment k by 4;

Emit(Lk:).

Emit(if _tn goto L(k+1)).

Emit(jump L(k+2)).

…

}

TAC for control structures

Let us now assume that we have figured our TAC *cgen()* functions for all Boolean operations.

Question: How would I encode an if statement in TAC code?

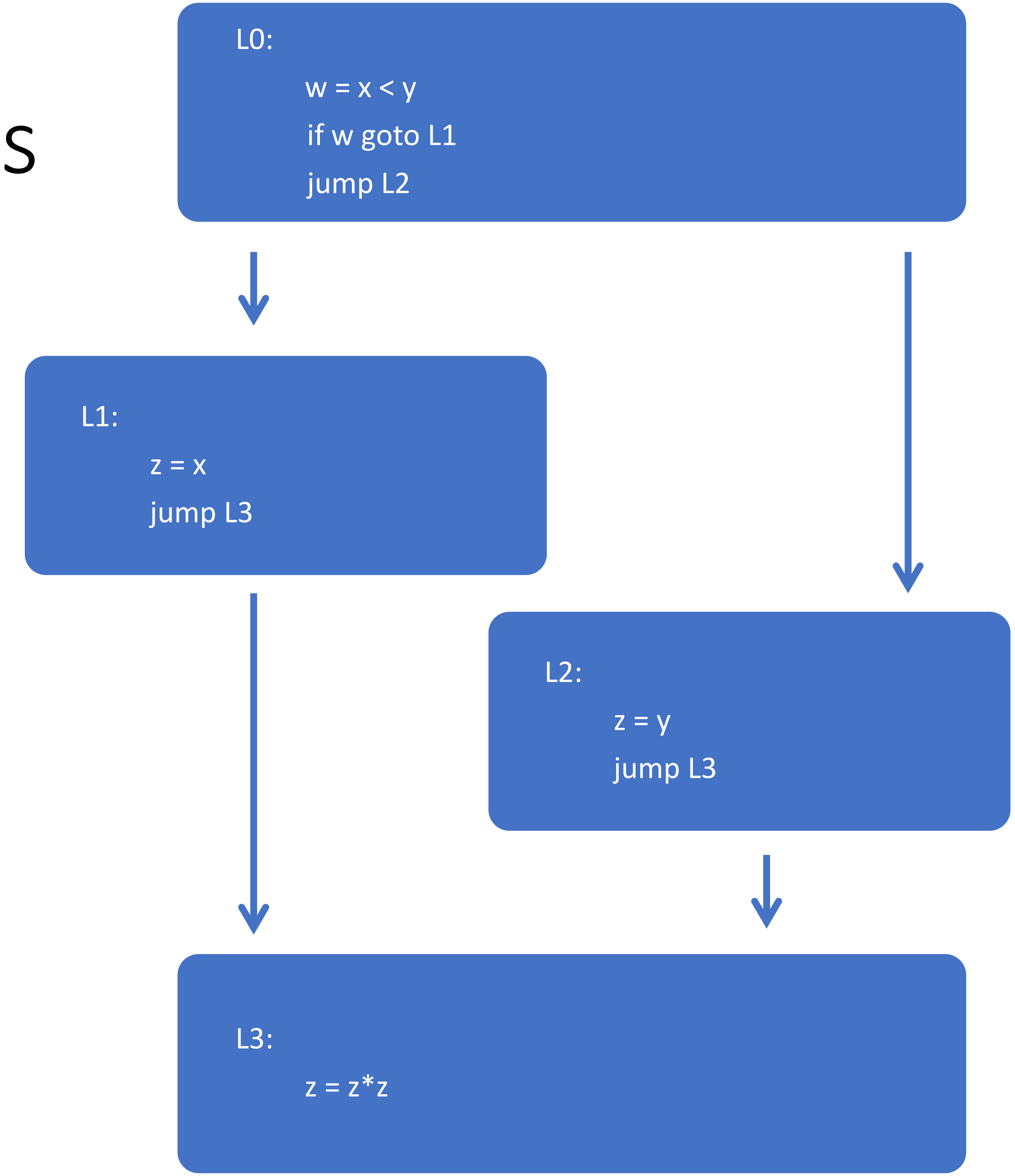
if(x < y)

    z = x;

else

    z = y;

*z = z*z;*



Restricted

TAC for control structures

TAC system counters: n, k.

cgen(if S1 S2 else S3) = {

Choose three new temporary t using our counter n. Increment n by 3;

Let _tn = cgen(S1);

Choose four new block labels Lk. Increment k by 4;

Emit(Lk:). Emit(if _tn goto L(k+1)). Emit(jump L(k+2)).

Emit(L(k+1):).

Let _t(n+1) = cgen(S2);

Emit(jump L(k+3)).

}

TAC for control structures

Let us now assume that we have figured our TAC *cgen()* functions for all Boolean operations.

Question: How would I encode an if statement in TAC code?

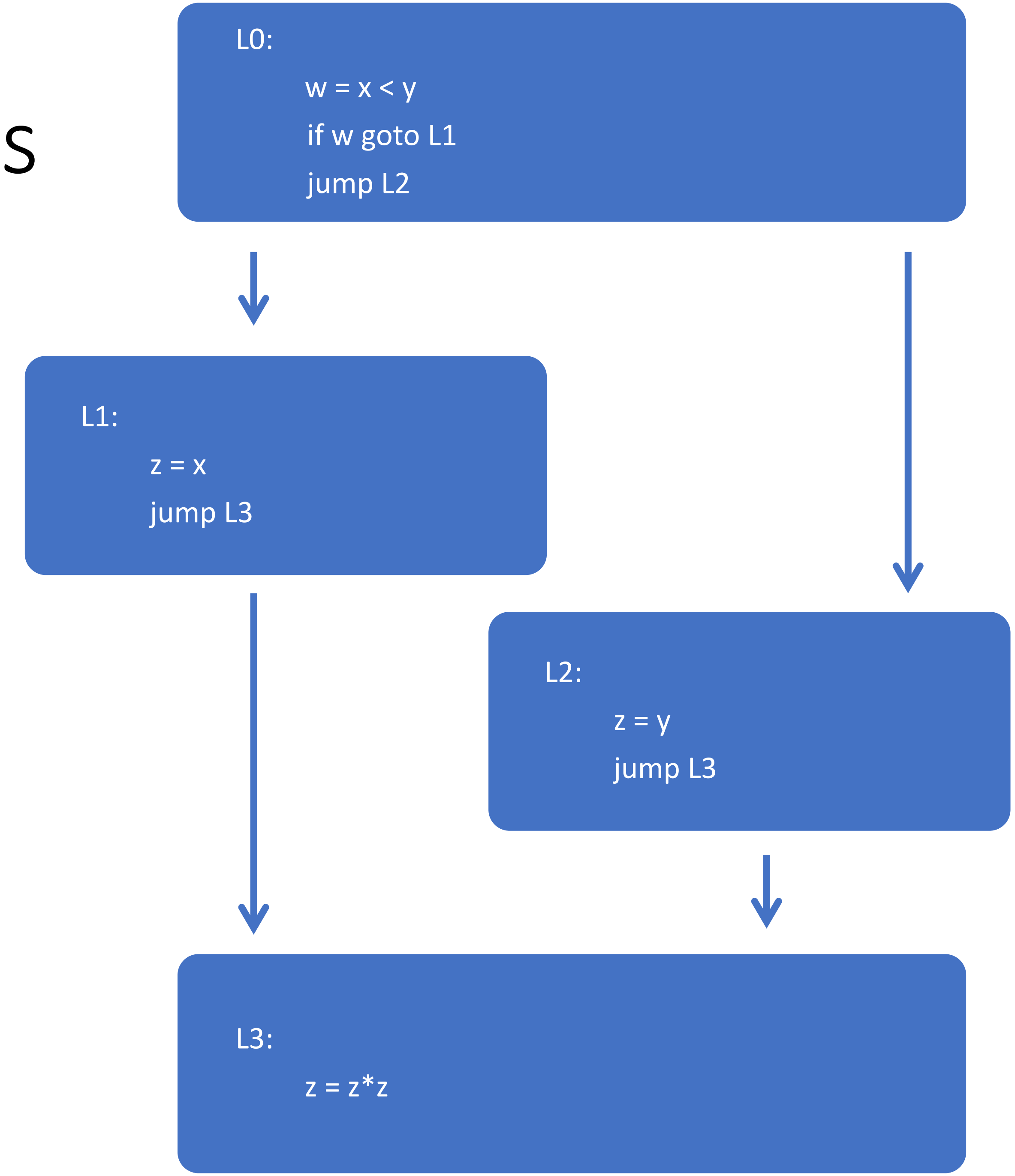
if(x < y)

z = x;

else

z = y;

*z = z*z;*



TAC for control structures

TAC system counters: n, k.

cgen(if S1 S2 else S3) = {

Choose three new temporary t using our counter n. Increment n by 3;

Let _tn = cgen(S1);

Choose four new block labels Lk. Increment k by 4;

Emit(Lk:). Emit(if _tn goto L(k+1)). Emit(jump L(k+2)).

Emit(L(k+1):). Let _t(n+1) = cgen(S2); Emit(jump L(k+3)).

Emit(L(k+2):). Let _t(n+2) = cgen(S3); Emit(jump L(k+3)).

}

TAC for control structures

Let us now assume that we have figured our TAC *cgen()* functions for all Boolean operations.

Question: How would I encode an if statement in TAC code?

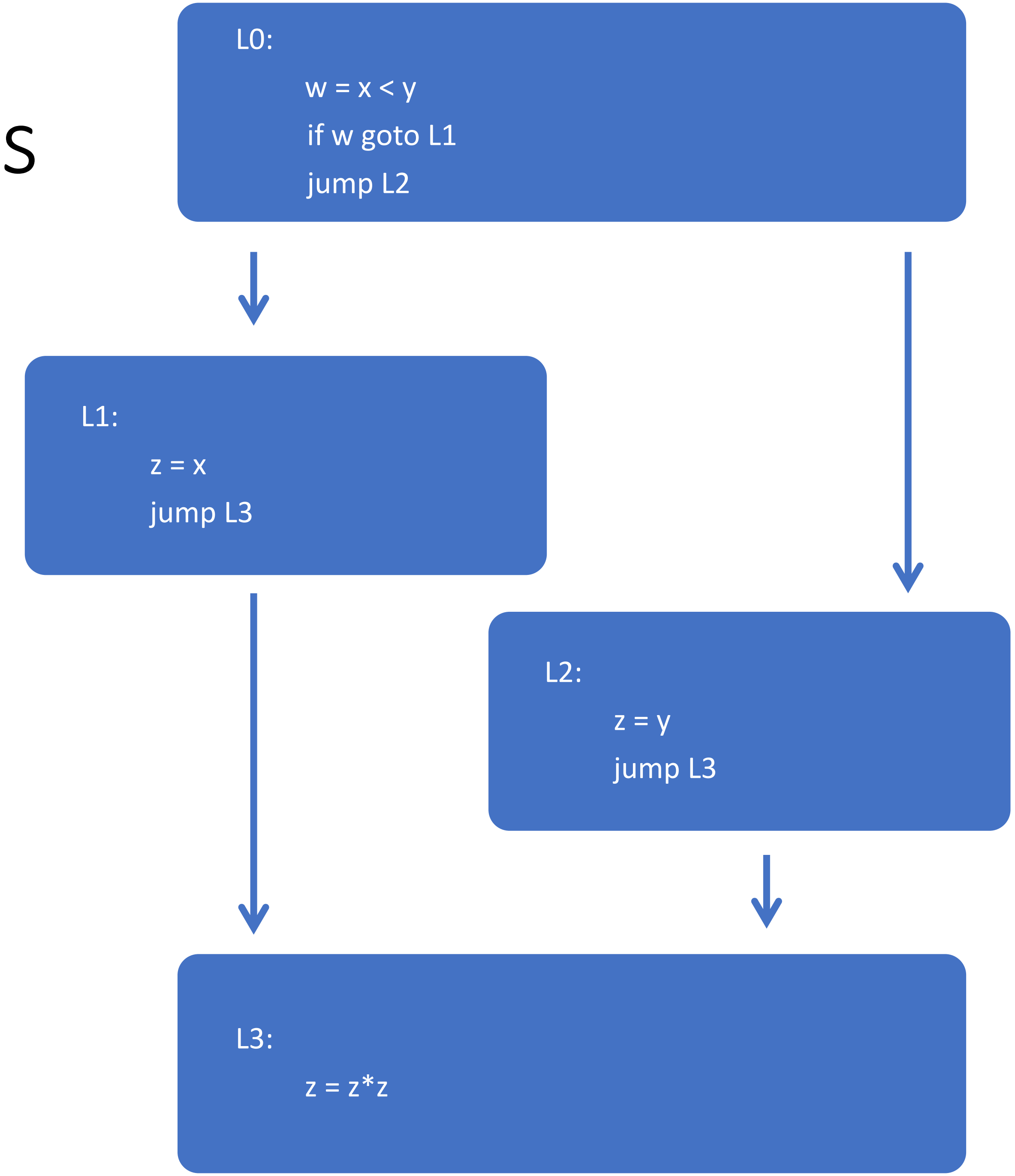
if(x < y)

    z = x;

else

    z = y;

*z = z*z;*



TAC for control structures

TAC system counters: n, k.

cgen(if S1 S2 else S3) = {

Choose three new temporary t using our counter n. Increment n by 3;

Let _tn = cgen(S1);

Choose four new block labels Lk. Increment k by 4;

Emit(Lk:). Emit(if _tn goto L(k+1)). Emit(jump L(k+2)).

Emit(L(k+1):). Let _t(n+1) = cgen(S2); Emit(jump L(k+3)).

Emit(L(k+2):). Let _t(n+2) = cgen(S3); Emit(jump L(k+3)).

Return?

}

TAC for control structures

TAC system counters: n, k.

cgen(if S1 S2 else S3) = {

Choose three new temporary t using our counter n. Increment n by 3;

Let _tn = cgen(S1);

Choose four new block labels Lk. Increment k by 4;

Emit(Lk:). Emit(if _tn goto L(k+1)). Emit(jump L(k+2)).

Emit(L(k+1):). Let _t(n+1) = cgen(S2); Emit(jump L(k+3)).

Emit(L(k+2):). Let _t(n+2) = cgen(S3); Emit(jump L(k+3)).

~~Return?~~

}

TAC for control structures

TAC system counters: n, k.

cgen(if S1 S2 else S3) = {

Choose three new temporary t using our counter n. Increment n by 3;

Let _tn = cgen(S1);

Choose four new block labels Lk. Increment k by 4;

Emit(Lk:). Emit(if _tn goto L(k+1)). Emit(jump L(k+2)).

Emit(L(k+1):). Let _t(n+1) = cgen(S2); Emit(jump L(k+3)).

Emit(L(k+2):). Let _t(n+2) = cgen(S3); Emit(jump L(k+3)).

Emit(L(k+3):).

}

TAC for if statement

Consider the code below. What will be the TAC for this code?

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

TAC for if statement

Consider the code below. What will be the TAC for this code?

if(x < y)

    z = x;

else

    z = y;

*z = z*z;*

We have to run two *cgen()* operation in sequence to get the full TAC code.

- *cgen(if x<y z=x else x=y)*
- *cgen(z = z*z)*

TAC for if statement

TAC system counters: n = 0, k = 0.

cgen(if S1 S2 else S3) = {

Choose three new temporary t using our counter n. Increment n by 3;

Let _tn = cgen(S1);

Choose four new block labels Lk. Increment k by 4;

Emit(Lk:). Emit(if _tn goto L(k+1)). Emit(jump L(k+2)).

Emit(L(k+1):). Let _t(n+1) = cgen(S2); Emit(jump L(k+3)).

Emit(L(k+2):). Let _t(n+2) = cgen(S3); Emit(jump L(k+3)).

Emit(L(k+3):).

}

TAC for if statement

TAC system counters: n = 0, k = 0.

cgen(if x<y z=x else x=y) = {

Choose three new temporary t using our counter n. Increment n by 3;

Let _tn = cgen(x<y);

Choose four new block labels Lk. Increment k by 4;

Emit(Lk:). Emit(if _tn goto L(k+1)). Emit(jump L(k+2)).

Emit(L(k+1):). Let _t(n+1) = cgen(z=x); Emit(jump L(k+3)).

Emit(L(k+2):). Let _t(n+2) = cgen(x=y); Emit(jump L(k+3)).

Emit(L(k+3):).

}

TAC for if statement

TAC system counters: n = 3, k = 0.

cgen(if x<y z=x else x=y) = {

Choose three new temporary t using our counter n. Increment n by 3;

Let _t0 = cgen(x<y);

Choose four new block labels Lk. Increment k by 4;

Emit(Lk:). Emit(if _t0 goto L(k+1)). Emit(jump L(k+2)).

Emit(L(k+1):). Let _t1 = cgen(z=x); Emit(jump L(k+3)).

Emit(L(k+2):). Let _t2 = cgen(x=y); Emit(jump L(k+3)).

Emit(L(k+3):).

}

TAC for if statement

TAC system counters: n = 3, k = 4.

cgen(if x<y z=x else x=y) = {

Choose three new temporary t using our counter n. Increment n by 3;

Let _t0 = cgen(x<y);

Choose four new block labels Lk. Increment k by 4;

Emit(L0:). Emit(if _t0 goto L1). Emit(jump L2).

Emit(L1:). Let _t1 = cgen(z=x); Emit(jump L3).

Emit(L2:). Let _t2 = cgen(x=y); Emit(jump L3).

Emit(L3:).

}

TAC for if statement

TAC system counters: n = 3, k = 4.

cgen(if x<y z=x else x=y) = {

Choose three new temporary t using our counter n. Increment n by 3;

Let _t0 = cgen(x<y); ←

Choose four new block labels Lk. Increment k by 4;

Emit(L0:). Emit(if _t0 goto L1). Emit(jump L2).

Emit(L1:). Let _t1 = cgen(z=x); Emit(jump L3).

Emit(L2:). Let _t2 = cgen(x=y); Emit(jump L3).

Emit(L3:).

}

TAC code

TAC for if statement

TAC system counters: n = 3, k = 4.

cgen(e1 < e2) = {

Choose three new temporary t using our counter n (we shall name them as _tn, _t(n+1), _t(n+2));

Increment n by 3;

Let _tn = cgen(e1);

Let _t(n+1) = cgen(e2);

Emit (_t(n+2) = _tn + _t(n+1));

Return _t(n+2);

}

TAC code

TAC for if statement

TAC system counters: n = 6, k = 4.

*cgen(**x** < **y**) = {*

*Choose three new temporary t using our counter n (we shall name them as **_t4**, **_t5**, **_t6**);*

Increment n by 3;

*Let **_t4** = cgen(**x**);*

*Let **_t5** = cgen(**y**);*

*Emit (**_t6** = **_t4** < **_t5**);*

*Return **_t6**;*

}

TAC code

TAC for if statement

TAC system counters: n = 7, k = 4.

*cgen(**x < y**) = {*

*Choose three new temporary t using our counter n (we shall name them as **_t4**, **_t5**, **_t6**);*

Increment n by 3;

*Let **_t4** = cgen(**x**);*

*Let **_t5** = cgen(**y**);*

*Emit (**_t6** = **_t4 < _t5**);*

*Return **_t6**;*

}

TAC code

_t7 := x

_t4 := _t7

TAC for if statement

TAC system counters: n = 8, k = 4.

*cgen(**x < y**) = {*

*Choose three new temporary t using our counter n (we shall name them as **_t4**, **_t5**, **_t6**);*

Increment n by 3;

*Let **_t4** = cgen(**x**);*

*Let **_t5** = cgen(**y**);*

*Emit (**_t6** = **_t4 < _t5**);*

*Return **_t6**;*

}

TAC code

_t7 := x
_t4 := _t7
_t8 := y
_t5 := _t8

TAC for if statement

TAC system counters: n = 8, k = 4.

*cgen(**x < y**) = {*

*Choose three new temporary t using our counter n (we shall name them as **_t4**, **_t5**, **_t6**);*

Increment n by 3;

*Let **_t4** = cgen(**x**);*

*Let **_t5** = cgen(**y**);*

*Emit (**_t6** = **_t4 < _t5**);*

*Return **_t6**;*

}

TAC code

```
_t7 := x
_t4 := _t7
_t8 := y
_t5 := _t8
_t6 := _t4 < _t5
```

TAC for if statement

TAC system counters: n = 3, k = 4.

cgen(if x<y z=x else x=y) = {

Choose three new temporary t using our counter n. Increment n by 3;

Let _t0 = cgen(x<y);

Choose four new block labels Lk. Increment k by 4;

Emit(L0:). Emit(if _t0 goto L1). Emit(jump L2).

Emit(L1:). Let _t1 = cgen(z=x); Emit(jump L3).

Emit(L2:). Let _t2 = cgen(x=y); Emit(jump L3).

Emit(L3:).

}

TAC code

```
_t7 := x
_t4 := _t7
_t8 := y
_t5 := _t8
_t6 := _t4 < _t5
_t0 := _t6
L0:
```

TAC for if statement

TAC system counters: n = 3, k = 4.

cgen(if x<y z=x else x=y) = {

Choose three new temporary t using our counter n. Increment n by 3;

Choose four new block labels Lk. Increment k by 4;

Emit(L0:). Let _t0 = cgen(x<y);

Emit(if _t0 goto L1). Emit(jump L2).

Emit(L1:). Let _t1 = cgen(z=x); Emit(jump L3).

Emit(L2:). Let _t2 = cgen(x=y); Emit(jump L3).

Emit(L3:).

}

TAC code

```
L0:
_t7 := x
_t4 := _t7
_t8 := y
_t5 := _t8
_t6 := _t4 < _t5
_t0 := _t6
```

Revised TAC for control structures

TAC system counters: n, k.

cgen(if S1 S2 else S3) = {

Choose three new temporary t using our counter n. Increment n by 3;

Choose four new block labels Lk. Increment k by 4;

Emit(Lk:). Let _tn = cgen(S1);

Emit(if _tn goto L(k+1)). Emit(jump L(k+2)).

Emit(L(k+1):). Let _t(n+1) = cgen(S2); Emit(jump L(k+3)).

Emit(L(k+2):). Let _t(n+2) = cgen(S3); Emit(jump L(k+3)).

Emit(L(k+3):).

}

TAC for if statement

TAC system counters: n = 8, k = 4.

cgen(if x<y z=x else x=y) = {

Choose three new temporary t using our counter n. Increment n by 3;

Choose four new block labels Lk. Increment k by 4;

Emit(L0:). Let _t0 = cgen(x<y);

Emit(if _t0 goto L1). Emit(jump L2).

Emit(L1:). Let _t1 = cgen(z=x); Emit(jump L3).

Emit(L2:). Let _t2 = cgen(x=y); Emit(jump L3).

Emit(L3:).

}

TAC code

```
L0:
_t7 := x
_t4 := _t7
_t8 := y
_t5 := _t8
_t6 := _t4 < _t5
_t0 := _t6
if _t0 goto L1
jump L2
```

TAC for if statement

TAC system counters: n = 8, k = 4.

cgen(if x<y z=x else x=y) = {

Choose three new temporary t using our counter n. Incremen

Choose four new block labels Lk. Increment k by 4;

Emit(L0:). Let _t0 = cgen(x<y);

Emit(if _t0 goto L1). Emit(jump L2).

Emit(L1:). Let _t1 = cgen(z=x); Emit(jump L3).

Emit(L2:). Let _t2 = cgen(x=y); Emit(jump L3).

Emit(L3:).

}

TAC code

```
L0:
_t7 := x
_t4 := _t7
_t8 := y
_t5 := _t8
_t6 := _t4 < _t5
_t0 := _t6
if _t0 goto L1
jump L2
L1:
z := x
_t1:= null
jump L3
```

TAC for if statement

TAC system counters: n = 8, k = 4.

cgen(if x<y z=x else x=y) = {

Choose three new temporary t using our counter n. Incremen

Choose four new block labels Lk. Increment k by 4;

Emit(L0:). Let _t0 = cgen(x<y);

Emit(if _t0 goto L1). Emit(jump L2).

Emit(L1:). Let _t1 = cgen(z=x); Emit(jump L3).

Emit(L2:). Let _t2 = cgen(x=y); Emit(jump L3).

Emit(L3:).

}

TAC code

```
L0:
_t7 := x
_t4 := _t7
_t8 := y
_t5 := _t8
_t6 := _t4 < _t5
_t0 := _t6
if _t0 goto L1
jump L2
L1:
z := x
_t1 := null
jump L3
L2:
x := y
_t2 := null
jump L3
```

TAC for if statement

TAC system counters: n = 8, k = 4.

cgen(if x<y z=x else x=y) = {

Choose three new temporary t using our counter n. Incremen

Choose four new block labels Lk. Increment k by 4;

Emit(L0:). Let _t0 = cgen(x<y);

Emit(if _t0 goto L1). Emit(jump L2).

Emit(L1:). Let _t1 = cgen(z=x); Emit(jump L3).

Emit(L2:). Let _t2 = cgen(x=y); Emit(jump L3).

Emit(L3:).

}

TAC code

```
L0:
_t7 := x
_t4 := _t7
_t8 := y
_t5 := _t8
_t6 := _t4 < _t5
_t0 := _t6
if _t0 goto L1
jump L2
L1:
z := x
_t1 := null
jump L3
L2:
x := y
_t2 := null
jump L3
L3:
```

TAC for if statement

TAC system counters: n = 11, k = 4.

*cgen(**z = z * z**) = {*

*Choose three new temporary t using our counter n (we shall name them as **_t8**, **_t9**, **_t10**);*

Increment n by 3;

*Let **_t8** = cgen(**z**);*

*Let **_t9** = cgen(**z**);*

*Emit (**_t10** = **_t8 * _t9**);*

*Emit (**z = _t10**)*

Return null;

}

We have to run two *cgen()* operation in sequence to get the full TAC code.

- *cgen(if x<y z=x else x=y)*
- ***cgen(z = z*z)***

Note: this second operation needs not to be decomposed but could simply combine several concepts from previous *cgen()* functions together.

TAC for if statement

TAC system counters: n = 11, k = 4.

*cgen(**z = z * z**) = {*

*Choose three new temporary t using our counter n (we shall name them as **_t8**, **_t9**, **_t10**);*

Increment n by 3;

*Let **_t8** = cgen(**z**);*

*Let **_t9** = cgen(**z**);*

*Emit (**_t10** = **_t8** * **_t9**);*

*Emit (**z** = **_t10**)*

Return null;

}

TAC code

```
L0:
_t7 := x
_t4 := _t7
_t8 := y
_t5 := _t8
_t6 := _t4 < _t5
_t0 := _t6
if _t0 goto L1
jump L2
L1:
z := x
_t1 := null
jump L3
L2:
x := y
_t2 := null
jump L3
L3:
_t8 := z
_t9 := z
_t10 = z*z
z := _t10
```

TAC for if statement

Consider the code below. What will be the TAC for this code?

if(x < y)

z = x;

else

z = y;

*z = z*z;*

TAC code

```
L0:
_t7 := x
_t4 := _t7
_t8 := y
_t5 := _t8
_t6 := _t4 < _t5
_t0 := _t6
if _t0 goto L1
jump L2
L1:
z := x
_t1 := null
jump L3
L2:
x := y
_t2 := null
jump L3
L3:
_t8 := z
_t9 := z
_t10 = z*z
z := _t10
```

TAC for if statement

Not exactly optimal, but again, we do not care!

A different part of the compiler will be in charge of optimizing this TAC code later on!

TAC code

```
L0:
_t7 := x
_t4 := _t7
_t8 := y
_t5 := _t8
_t6 := _t4 < _t5
_t0 := _t6
if _t0 goto L1
jump L2
L1:
z := x
_t1 := null
jump L3
L2:
x := y
_t2 := null
jump L3
L3:
_t8 := z
_t9 := z
_t10 = z*z
z := _t10
```

TAC for function calls

- Writing TAC becomes a bit tricky when accounting for function calls.
- For instance, consider the code below.

```
void f(int z) {
    int x;
    x = z*z
    return x;
}
void main(){
    f(11);
    int y = f(8);
}
```

TAC for function calls

The code could be transformed into a TAC using our *cgen()*, as shown on the right.

```
void f(int z) {
    int x;
    x = z*z
    return x;
}
void main(){
    f(11);
    int y = f(8);
}
```

TAC code

L0:
_t0 := z
_t1 := _t0*_t0
Return _t1? (not allowed in TAC)

L1:
_t2 := 11
Goto L0 with _t2 as z?
Nothing to catch the return of that function?
_t3 := 8
Goto L0 with _t3 as z?
Something to catch the return of that function?

Handling function calls with a stack

What is missing here?

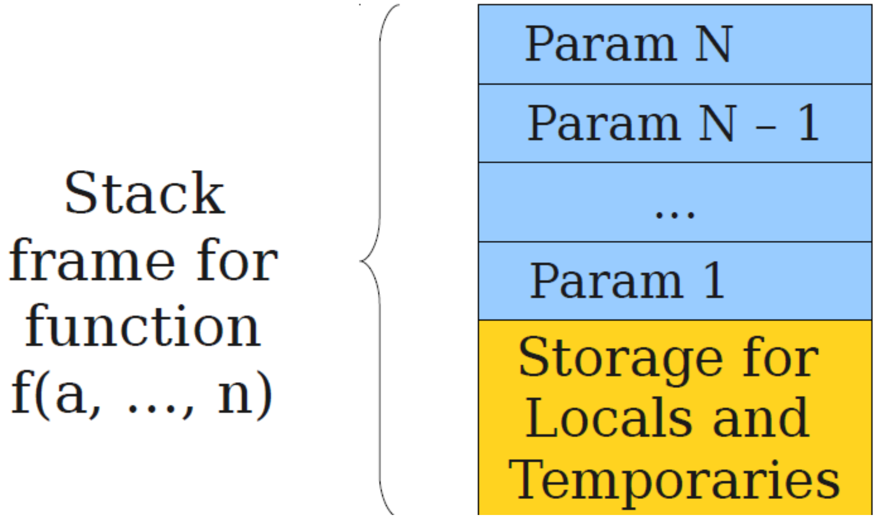
- Need to find a way to transfer parameters (x1, …, xn) from current scope (e.g. the main() one) to a function scope (e.g. function f called in main()).
- We need to know where the TAC code for f() is.
- In case of a return in f(), need a way to retrieve parameters from the function scope and bring them back into the enclosing scope.
- Should be done in a way to prevent clashes
- Should cover for null returns
- Should cover for returns not being caught, etc.

Handling function calls with a stack

Idea: Use a stack to represent the scope of a given function.

Scope will decompose in two parts

- The parameters passed to this function (could be none).
- The temporary parameters (_tn) used by this function during the computation.
- In this second case, we need to know the number of _tn used by the function f!

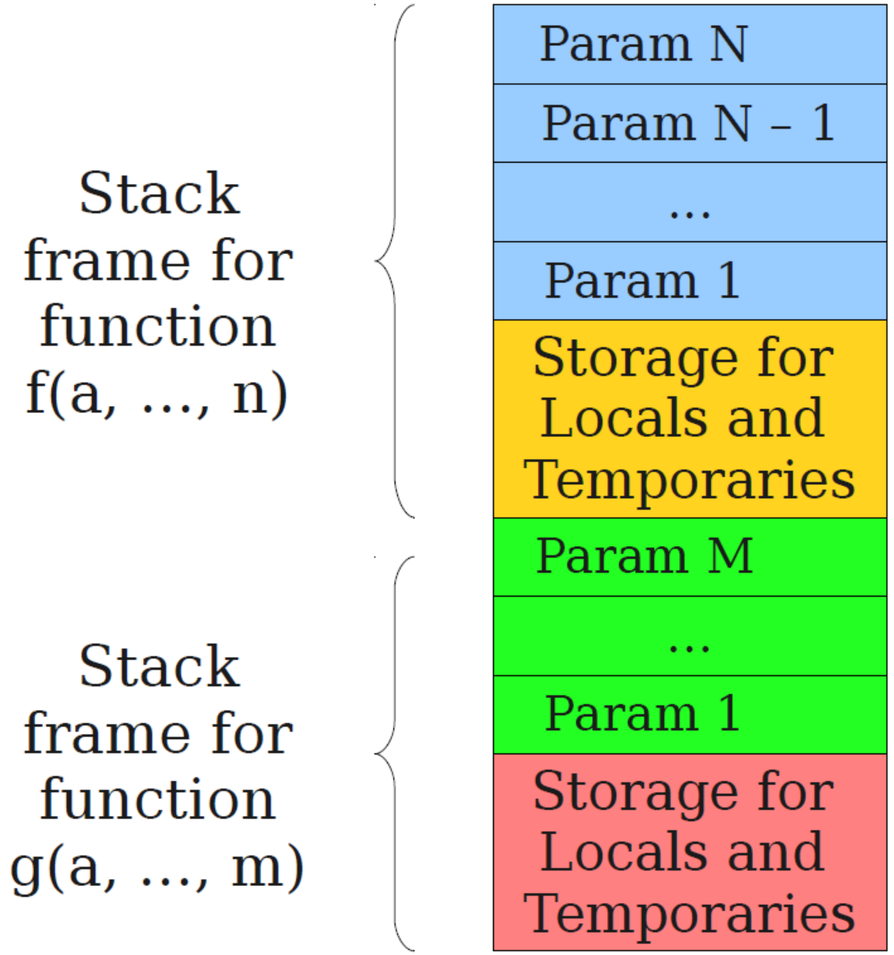


Handling function calls with a stack

Idea: Use a stack to represent the scope of a given function.

Whenever a function g() is called withing the scope of function f(), we will:

- Push some parameters from the scope of f() into the parameters stack of g().
- Allocate space for the temporary variables used by g().

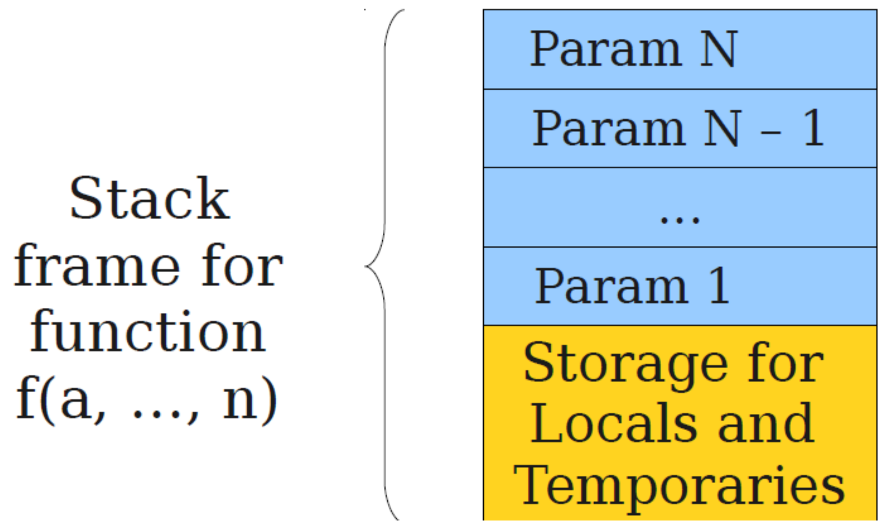


Handling function calls with a stack

Idea: Use a stack to represent the scope of a given function.

Upon completing g(), we will

- Pop some values from the temporary variables and add them to the temporaries of f().
- Or not (if no return specified).



Handling function calls with a stack

TAC can be used to handle function calls in a similar way to how it handles other code constructs.

- When a function call is encountered in the code, the TAC generator will typically create a new label for the function and generate code to push any arguments onto the stack.
- We denote this operation "*push _tn*" in TAC.

Handling function calls with a stack

TAC can be used to handle function calls in a similar way to how it handles other code constructs.

- When a function call is encountered in the code, the TAC generator will typically create a new label for the function and generate code to push any arguments onto the stack.
- In addition, the TAC needs to allocate space into the stack for all the temporary variables that function f() will require.
- It does so, with a command called “BeginFunc N”, which allocates N bytes in the stack to store data.
- In general memory entries are 32 bytes, so N is expected to be a multiple of 4 (e.g. N = 16 → 4 temporary variables).

Handling function calls with a stack

TAC can be used to handle function calls in a similar way to how it handles other code constructs.

- When the function returns, the TAC generator will generate code to pop any return values off the stack and store them in the appropriate variables.

It is done in two steps.

- The first step, reclaims the parameters space we opened with “push” operations, and this is done with a TAC command called “EndFunc”.

Handling function calls with a stack

TAC can be used to handle function calls in a similar way to how it handles other code constructs.

- When the function returns, the TAC generator will generate code to pop any return values off the stack and store them in the appropriate variables.

It is done in two steps.

- The second step, retrieves values that must be returned (and clears the rest of the temporary parameters in the process).
- This is done with a “pop N” operations, which allows the callee f() to retrieve parameters N bytes of data from the temporary variables in the scope of g().
- Basically implementing a return.

A blast from the past!

In a sense, what we are doing here is very similar to the way we handled **Stacks and Procedures** in **50.002 Computation Structures**!

Need a refresher? https://natalieagus.github.io/50002/notes/stackandprocedures

50.002 Computation Structures
Information Systems Technology and Design
Singapore University of Technology and Design

Stack and Procedures

You can find the lecture video here. You can also **click** on each header to bring you to the section of the video covering the subtopic.

Overview

In the previous chapter, we learned the basics of how to naively compile C-language into β assembly language. β UASM provides a layer of abstraction such that we don't need to bother ourselves with the details on how to load each and every bytes of instruction onto the memory unit, or keeping up with accounting matters such as physical memory addresses (we can replace these with *labels* instead).

In this chapter, we will learn about **function call procedures**, and why we need to understand another concept called the *stacks*. Both will allow us to have **reusable** code fragments which we normally know as **functions** that we can **call** as needed.

TAC for function calls

Using previous concepts, we have the TAC code for the function below.

```
void f(int z) {
    int x;
    x = z*z
    return x;
}
void main(){
    f(11);
    int y = f(8);
}
```

TAC code

```
L0:
BeginFunc 8
_t0 := z
_t1 := _t0*_t0
EndFunc

L1:
BeginFunc 12
_t2 := 11
Push _t2
LCall L0
Pop 4
_t3 := 8
Push _t3
LCall L0
y := Pop 4
EndFunc
```

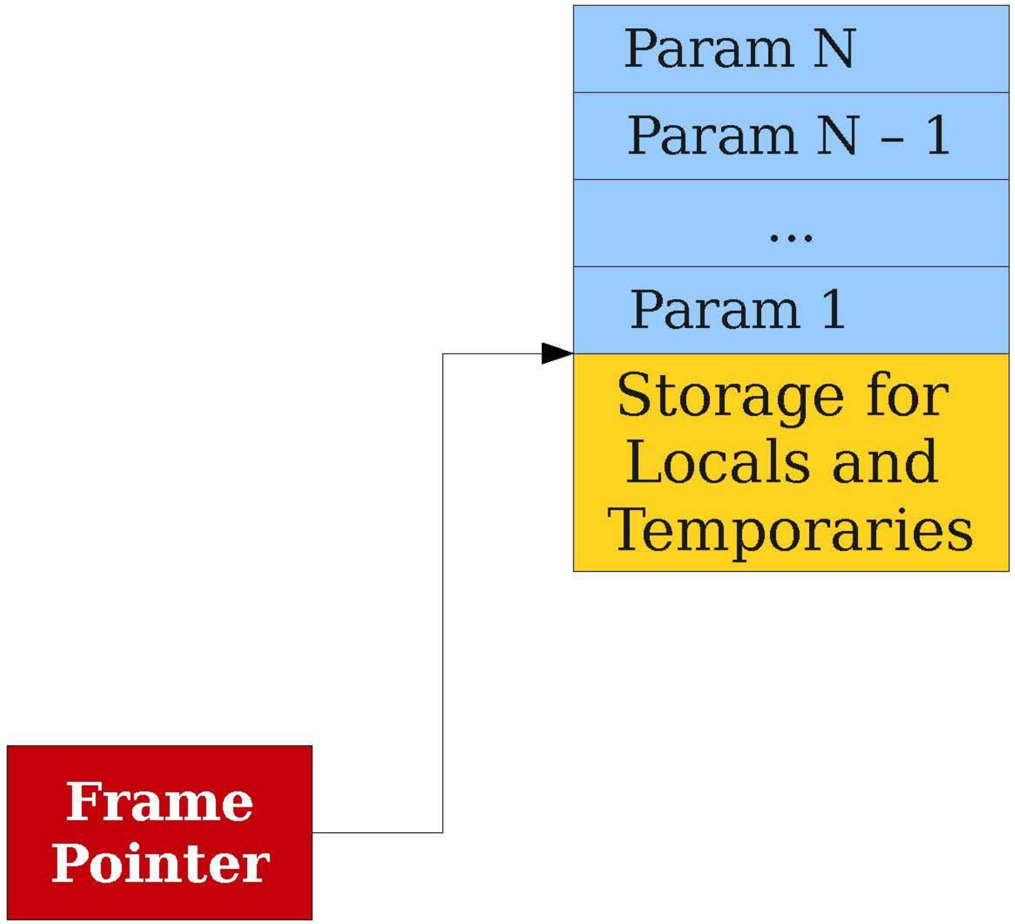
Program counter, Frame pointers, Global variables, etc.

In general, TAC does not specify where variables and temporaries are stored in the stack. It might require to bring in concepts from Stacks and Procedures, such as:

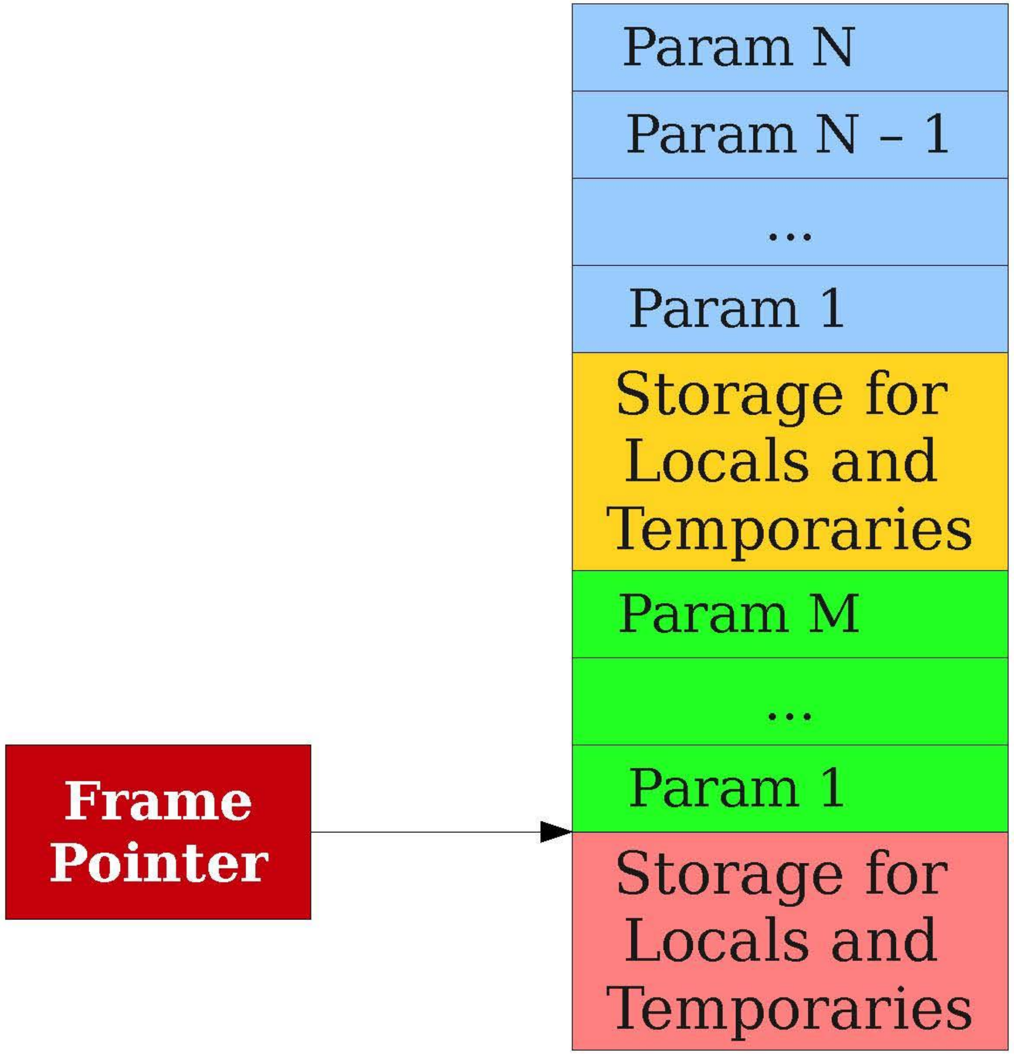
- **Frame Pointer/Base pointer:** A pointer points to the base of the current function activation record on the stack.

The activation record typically includes the function's local variables, any temporary values or registers, and other metadata that is needed to execute the function.

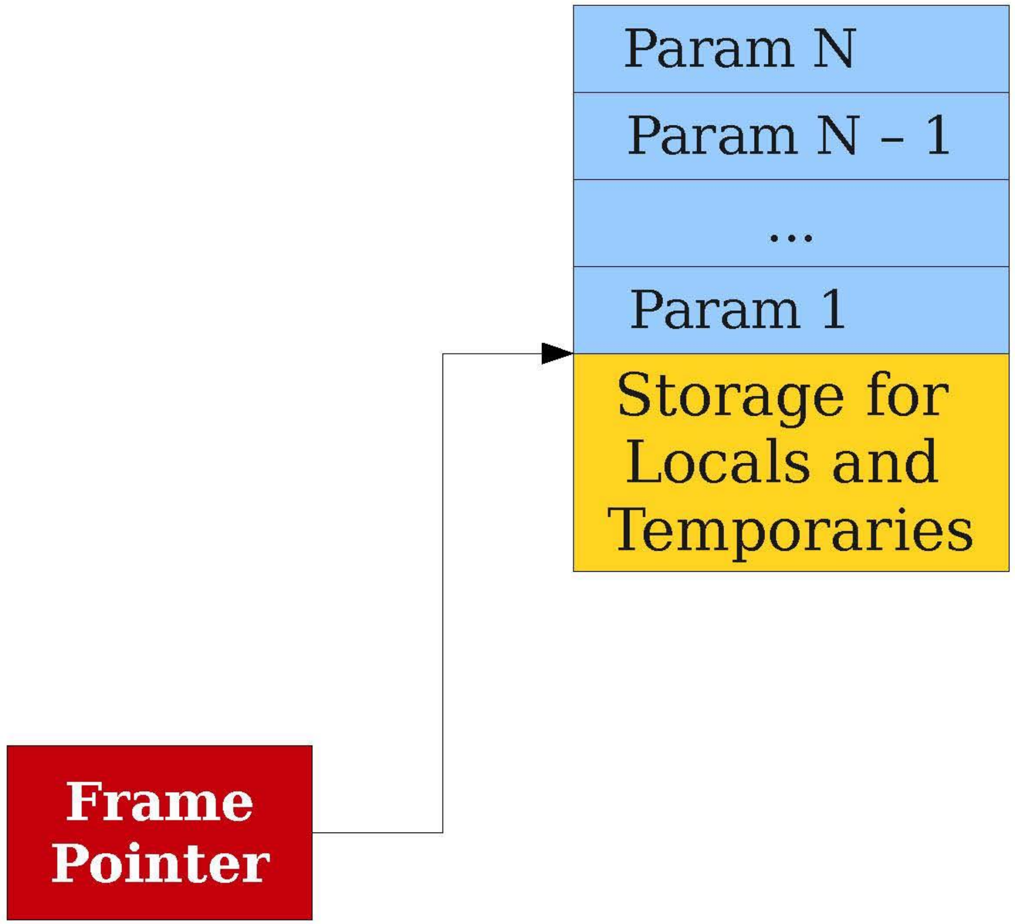
Using a frame pointer



Using a frame pointer



Using a frame pointer

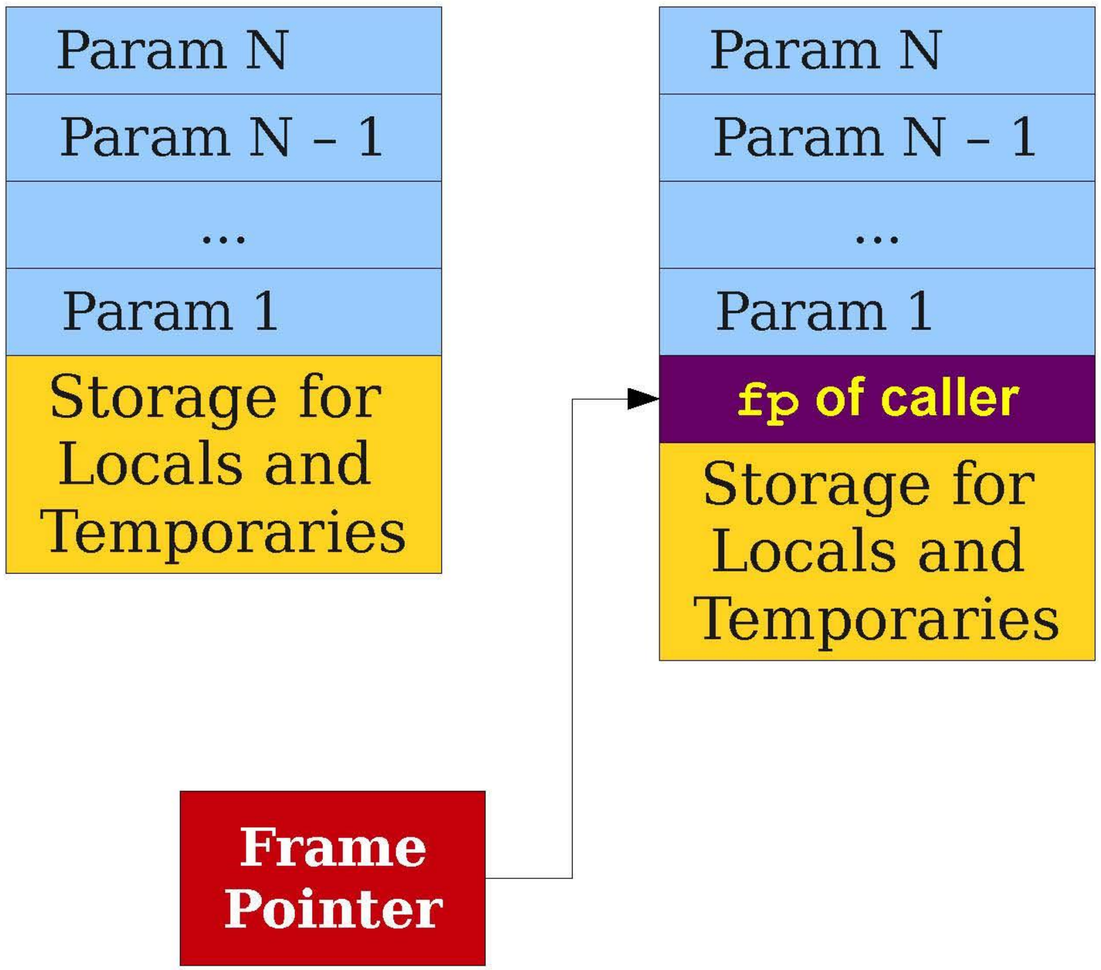


Program counter, Frame pointers, Global variables, etc.

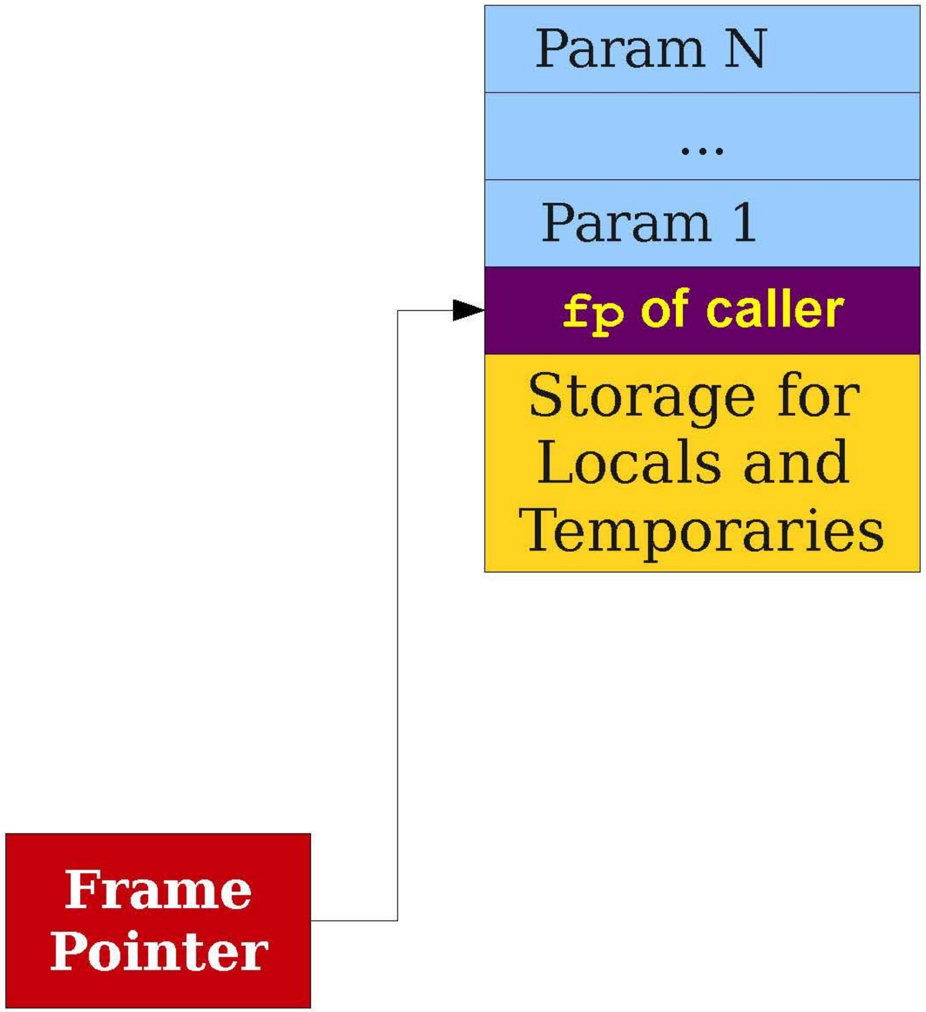
In general, TAC does not specify where variables and temporaries are stored in the stack. It might require to bring in concepts from Stacks and Procedures, such as:

- **Program Counter/Pointer:** Internally, the processor has a special register called the program counter/pointer (PC) that stores the address of the next instruction to execute.
- Whenever a function returns it shall restore the PC to the correct value, so that the calling function resumes its execution where it left off.
- We often store this address in our stack, along with function parameters and temporaries, in a location called the **Return Address**.

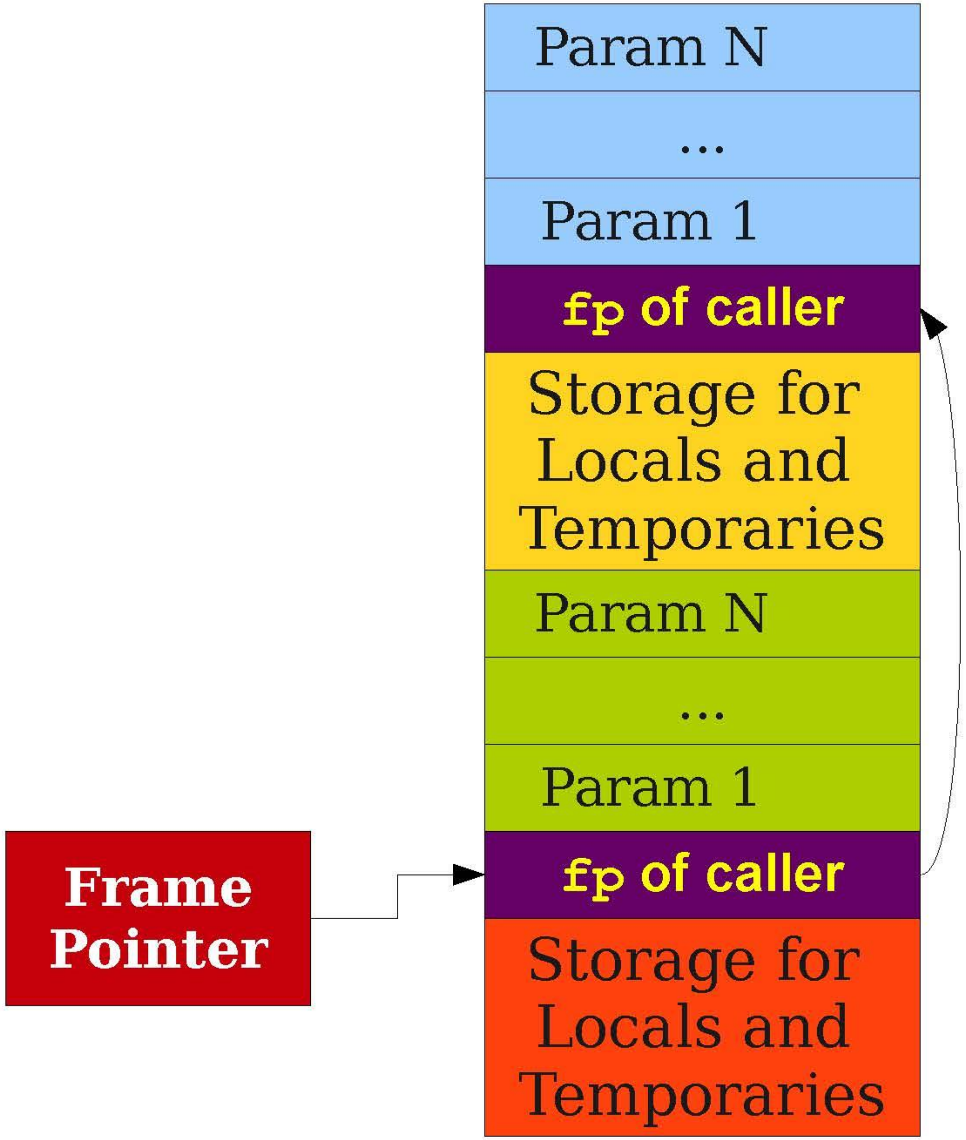
Using a caller addresses and return addresses



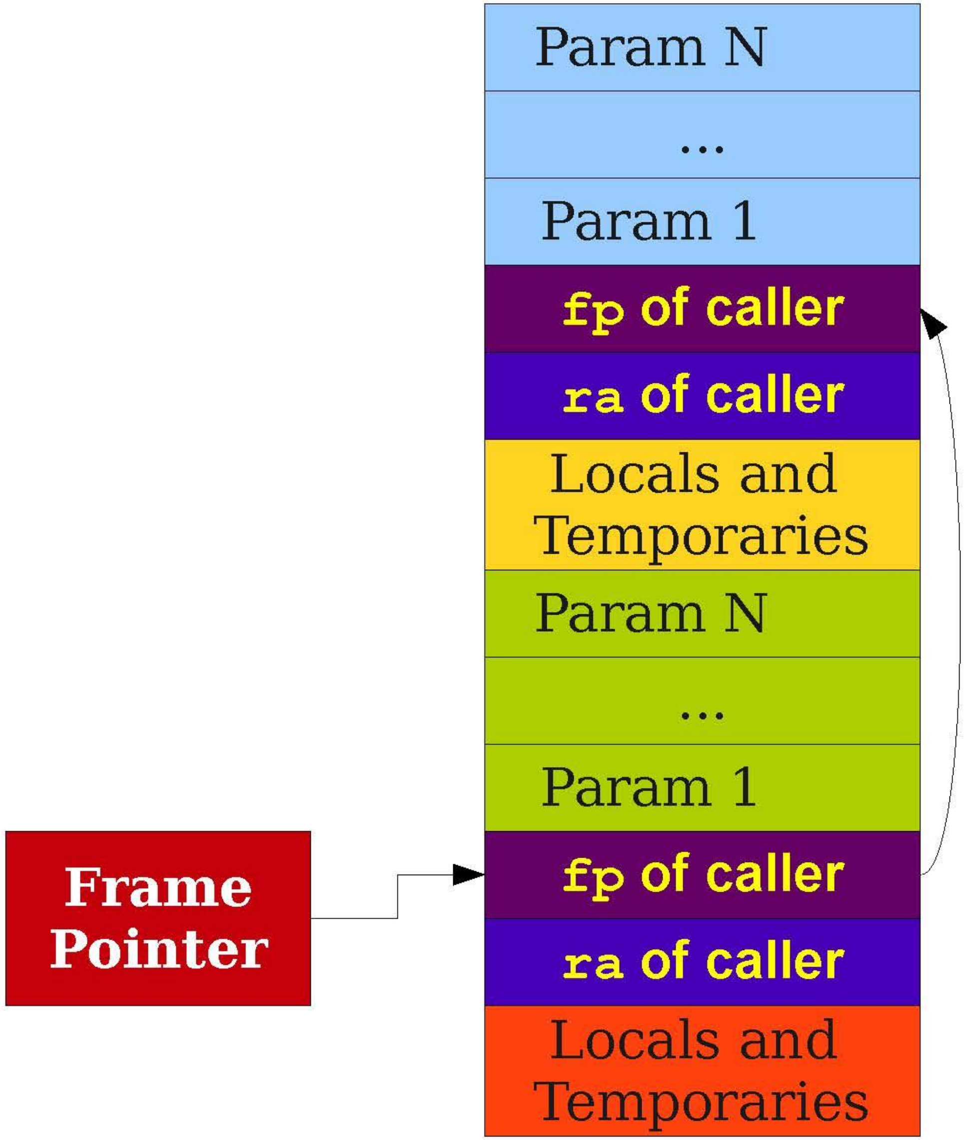
Using a caller addresses and return addresses



Using a caller addresses and return addresses



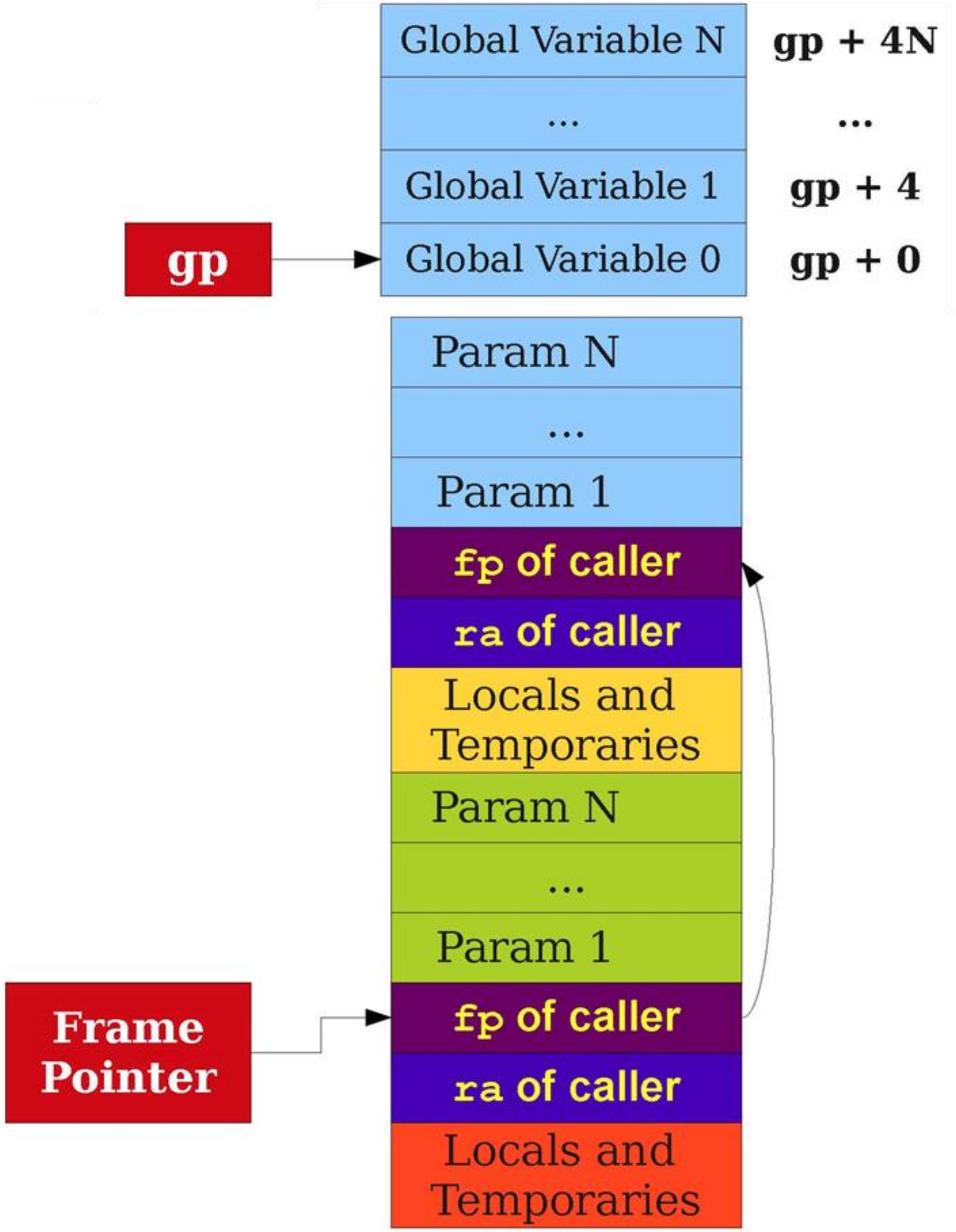
Using a caller addresses and return addresses



Program counter, Frame pointers, Global variables, etc.

In general, TAC does not specify where variables and temporaries are stored in the stack. It might require to bring in concepts from Stacks and Procedures, such as:

- **Global Pointer:** Some compilers will typically allow for LEGB scope searching, which means there should be a register for global variables that the TAC can use for computation.
- **A bit more advanced (and definitely out of scope), but often stored on top of the previous stack.**
- **Adding entries to global scope makes the stack go up.**
- **Opening function scopes makes the stack go down.**


Global Variable N
gp + 4N
...
...
Global Variable 1
gp + 4
gp
Global Variable 0
gp + 0
Param N
...
Param 1
fp of caller
ra of caller
Locals and Temporaries
Param N
...
Param 1
Frame Pointer
fp of caller
ra of caller
Locals and Temporaries

Program counter, Frame pointers, Global variables, etc.

In general, TAC does not specify where variables and temporaries are stored in the stack. It might require to bring in concepts from Stacks and Procedures, such as:

- Frame Pointer/Base Pointer,
- Program Counter/Pointer, and Return Addresses
- Global Pointer,
- Etc.

Out of scope for 50.051, but not 50.002!

Again, something already discussed in **Stacks and Procedures**, **50.002 Computation Structures**!
https://natalieagus.github.io/50002/notes/stackandprocedures

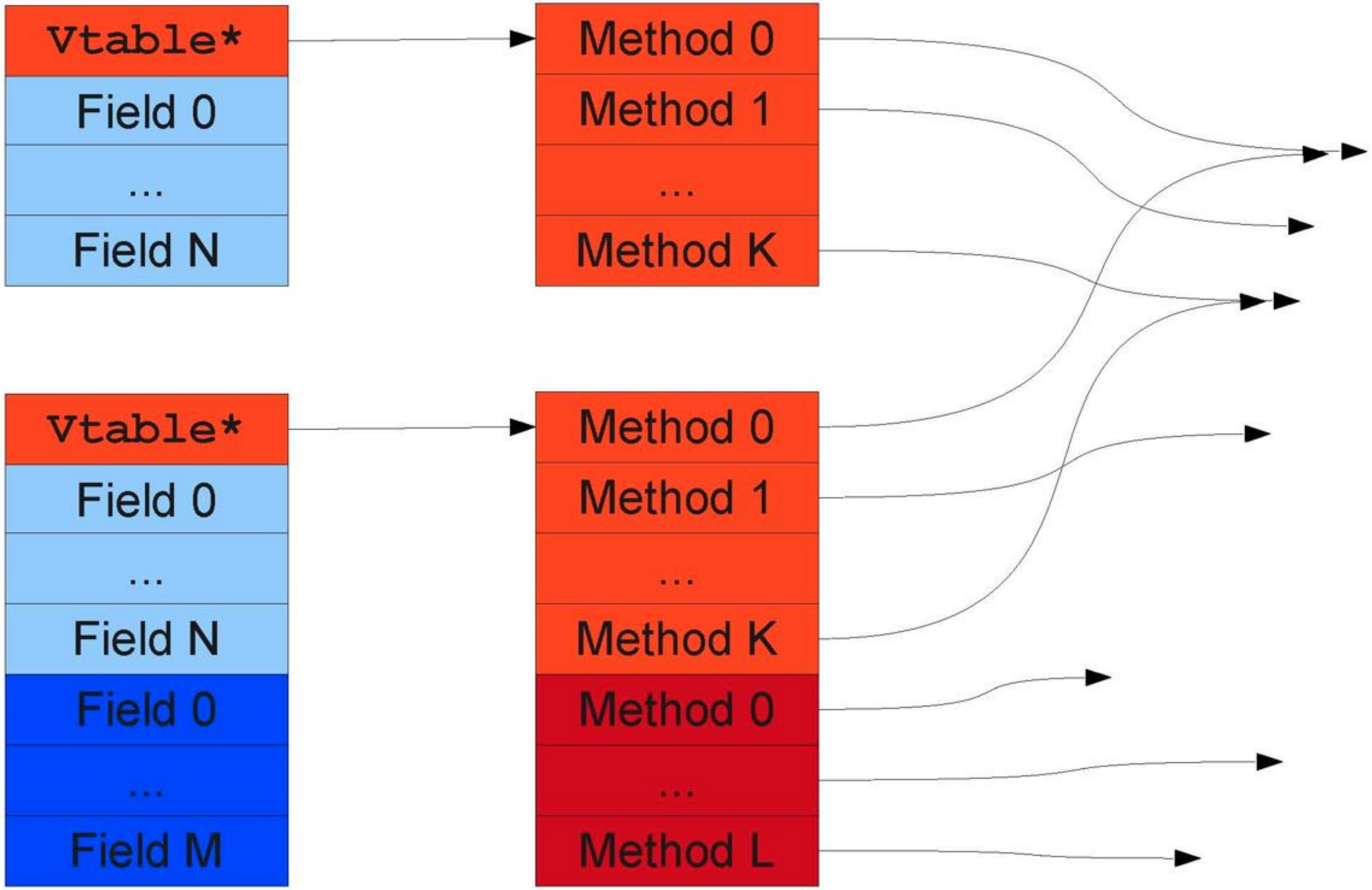
A quick word on TAC for objects

Encoding custom objects in TAC can prove to be very challenging.

- Need to allocate the right amount of space for attributes,
- Correctly link all methods and encode them correctly,
- Account for possible inheritance and overloading,
- Etc.
- Intuitively, we understand that it can be done with a vtable of some sort

Definitely out-of-scope!

A quick word on TAC for objects



Conclusion

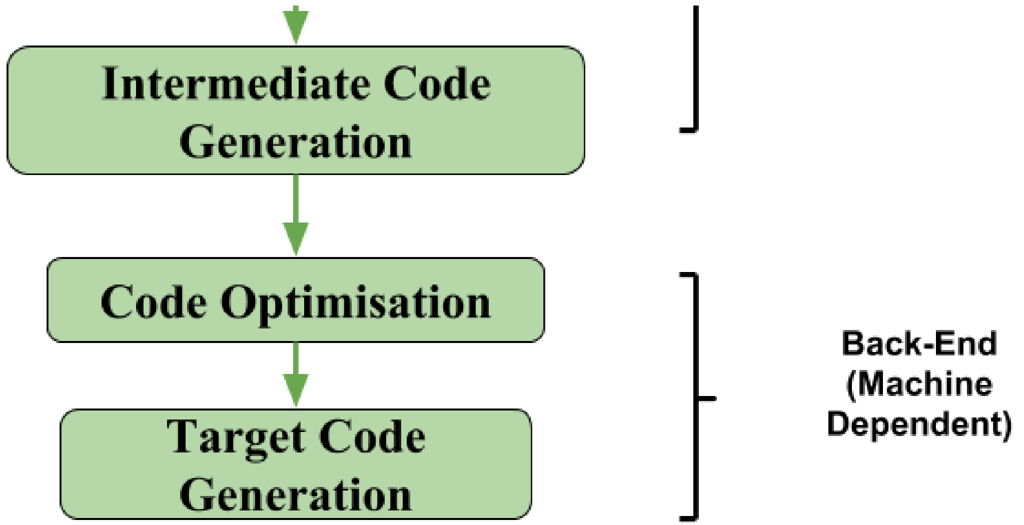
We understand how TAC is done and what intricacies might be required…

- Introduction to TAC.
- TAC for simple expressions.
- TAC for functions and function calls.
- TAC for objects.
- TAC for arrays.
- Generating TAC.
- A few low-level details.

The middle-end of a compiler

Our next step after Intermediate code generation is then Code optimization!

(Because, yes, our TAC codes at the moment are functional but not exactly looking great…)



TAC code

```
L0:
_t7 := x
_t4 := _t7
_t8 := y
_t5 := _t8
_t6 := _t4 < _t5
_t0 := _t6
if _t0 goto L1
jump L2
L1:
z := x
_t1 := null
jump L3
L2:
x := y
_t2 := null
jump L3
L3:
_t8 := z
_t9 := z
_t10 = z*z
z := _t10
```

Quiz time!

Which of the following best describes the structure of a three-address code?

A. Three operands and one operator

B. Two operands and two operators

C. One operand and three operators

D. Three operands and no operator

Quiz time!

Which of the following best describes the structure of a three-address code?

A. **Three operands and one operator**

B. Two operands and two operators

C. One operand and three operators

D. Three operands and no operator

Quiz time!

In the context of three-address code, what is the purpose of the "basic block" concept?

A. To define a sequence of code with a single entry and exit point
B. To divide code into smaller, manageable units for optimization
C. To handle loops and conditional statements correctly
D. All of the above
E. Non of the above

Quiz time!

In the context of three-address code, what is the purpose of the "basic block" concept?

A. To define a sequence of code with a single entry and exit point

B. To divide code into smaller, manageable units for optimization

C. To handle loops and conditional statements correctly

D. All of the above (more on B in the next lecture!)

E. Non of the above

Quiz time!

Which of the following is NOT a common operation in three-address code?

A. Arithmetic operations
B. Conditional jumps
C. Function calls
D. Dynamic memory allocation

Quiz time!

Which of the following is NOT a common operation in three-address code?

A. Arithmetic operations

B. Conditional jumps

C. Function calls

D. Dynamic memory allocation (that is something the backend will have to resolve later on during register allocation!)

Practice 2: a while statement TAC

Consider the code below.

```
x = 1
y = 3
while(x < y):
    x = x + 2;
    z = x;
z = z*z;
```

Question #1: How would you write the *cgen()* for the while statement?

You may look for inspiration by looking at the if() statement one.

Question #2: What would the TAC code be then?

Practice 3: a while statement TAC

Question #1: How would you write the *cgen()* function for the for loop statement in C?

You may look for inspiration by looking at the if() statement one and the while() statement in the previous activity.

Question #2 (somewhat difficult): How would you then agreement it to account for a possible break keyword?

Practice 4: a switch statement TAC

Question (somewhat tedious): How would you write the *cgen()* function for the switch statement in C?