50.051 Programming Language Concepts

W12-S1 Optimization and Middle-End

Matthieu De Mari

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

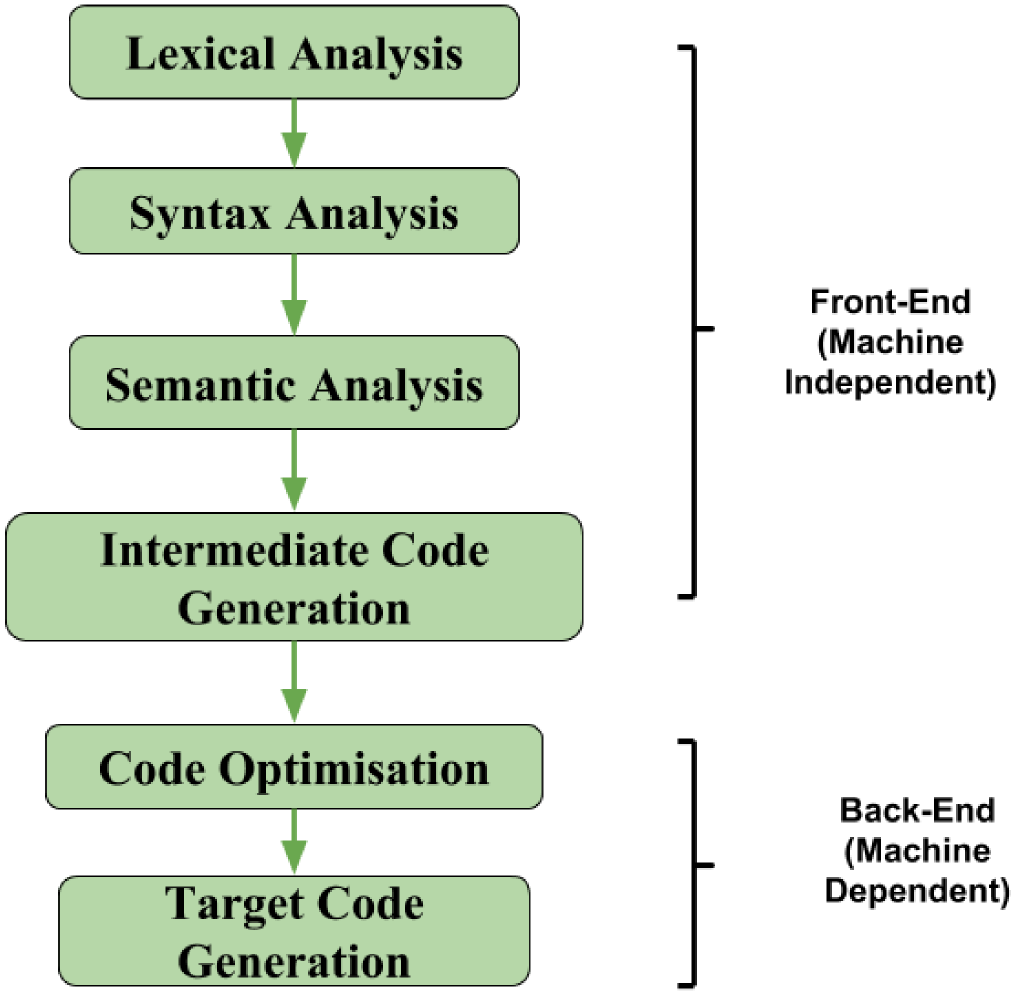
The middle-end of a compiler

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**.



Last lecture

During the last lecture, we

- Found a way to use the Abstract Syntax Tree representation of our source code after syntax analysis,
- Came up with the idea of using a translation mapping, that would translate any basic operation into its equivalent Three-Address Code (TAC) representation.
- Worked for basic arithmetic/boolean operations, control structures, loops, functions calls, etc.

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:
…
```

Last lecture

Problem: While this TAC code might be valid in terms of operations and expected outcomes, **it is clearly not optimal!**

- Many **redundancies**,
- Some **variables are not getting values**,
- Some **variables might have been assigned but are never used** (**dead code**).
- Etc.

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:
z := y
_t2 := null
jump L3
L3:
...
```

Restricted

Last lecture

Objective for IR optimization/middle-end: Improve the IR generated by the previous translation step to take better advantage of resources (memory, computational, etc.).

- A very important and complex topic, actively researched at the moment.
- Often NP-hard (or worse).
- Optimization techniques could be **local** or **global**.
- Subject to an **optimization trade-off**.

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:
z := y
_t2 := null
jump L3
L3:
...
```

Local optimization

Definition (local optimization):

In middle-end compilers, we call a **local optimization**, any technique that attempts to **optimize and modify the code contained in a single basic block of TAC code, without looking at the rest of the code**.

Typical local optimization techniques include:

- **Common subexpression elimination,**
- **Copy propagation,**
- **Dead code elimination,**
- Etc.

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.

Why basic blocks are nice

The only way to find out what a program will actually do is to run it.

Problems:

- The program might not terminate.
- The program might have some behaviour we did not see when we ran it on a particular input.

However, this is not a problem inside a basic block.

- Basic blocks contain no loops.
- There is only one path through the basic block.
- Easy to optimize and modify without messing it up!

Global/Interprocedural optimization

Definition (Global and interprocedural optimization):

In middle-end compilers, we call a **global optimization**, any technique that attempts to **optimize and modify some code contained in a single basic block of TAC code, using information gathered by looking at the rest of the code and other basic blocks**.

Usually a bit more advanced, as it requires to analyse the entire **control-flow graph** of the TAC code.

The optimization technique is even called **multiprocedural** if it looks at multiple control-flow graphs of multiple functions.

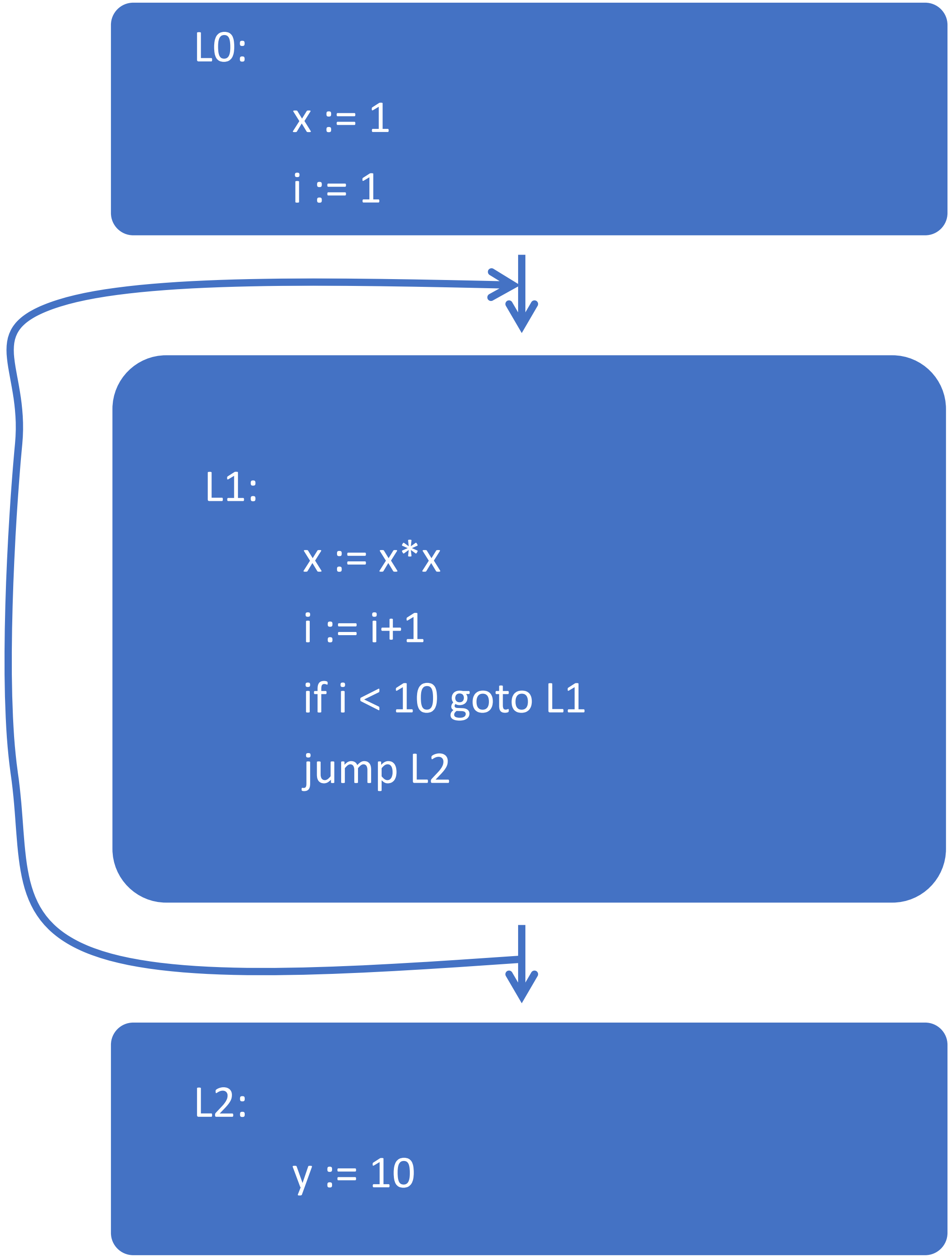
We will briefly discuss it, but most likely <u>out-of-scope</u>.

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



An important note before we start

Definition (semantics-preserving optimization):

When optimizing the TAC code, we want to **implement optimizations that preserve the semantics of an original program (i.e. semantics-preserving optimization)**.

For instance, we are okay with the idea of removing dead code, redundancy, as they would not change the logic of the source code per se (i.e. would preserve the original semantics).

Optimization techniques that would replace a bubble sort with a quicksort in our code, would not be considered as **semantics-preserving optimization** techniques. These could be interesting to study, but they are also out-of-scope.

Three important ideas

Idea #1: The term optimization implies looking for an “optimal” piece of code that could replace a given source program, and produce the exact same outcome.

- This is, in general, **undecidable** and falls in the category of **algorithmically unprovable problems**.
- Prefer to approach it heuristically, using some techniques that have been proved as simplifying the code, rather that optimizing it.
- Technically, we should rather refer to this step as “IR improvement” rather than “IR optimization”.

Three important ideas

Idea #2: What are we even optimizing?

There are many possible parameters we could consider when optimizing, but one that is terrible for sure is: “an optimal TAC code should use the fewest number of lines as possible”.

Instead, we will often look at a combination of parameters:

- **Runtime:** how long would it take to run the TAC code.
- **Memory:** how many temporary variables do you need for that code?
- **Power consumption:** not all instructions are equal, use the ones that are simpler (e.g. dividing an integer by 4 is equivalent to bitshift by 2 but the latter is far simpler to implement and less costly).

Three important ideas

Idea #3: How optimal do we even want the code to be anyway?

More specifically, how long do we want to spend optimizing the code before running it?

Question: Does it make sense to spend 1 minute compiling and optimizing the code if it makes use gain 0.1sec of execution?

- Maybe it does (if code is going to be executed a million times, or if optimization leads to drastic improvement in memory consumption).
- Maybe it does not (if you are prototyping an algorithm and using it once or twice only).

There is a **trade-off** to be found between the time spent optimizing and the gains we could hope to obtain anyway. Flags could be used even?

Restricted

Common Subexpression Elimination

Definition (Common Subexpression Elimination):

If, in the same basic block, we have two variable assignments

v1 = a op b

…

v2 = a op b

And the values of v1, a and b have not changed between the declarations of v1 and v2, we can perform **common subexpression elimination (CSE)** and rewrite the v2 assignment as

v2 = v1

It eliminates useless calculation and paves the way for more optimization.

Common Subexpression Elimination

Consider the code below, which needs common subexpression elimination.

$$x = a + b$$

$$y = a * c$$

$$z = a + b$$

$$w = a * c$$

$$t = x + z$$

Common Subexpression Elimination

Consider the code below, which needs common subexpression elimination.

$$\boldsymbol{x = a + b}$$

$$y = a * c$$

$$\boldsymbol{z = a + b}$$

$$w = a * c$$

$$t = x + z$$

Step 1: Recognize that x = a + b and z = a + b are two candidate expressions that could require common subexpression elimination.

Common Subexpression Elimination

Consider the code below, which needs common subexpression elimination.

$$\boldsymbol{x = a + b}$$

$$y = a * c$$

$$\boldsymbol{z = a + b}$$

$$w = a * c$$

$$t = x + z$$

Step 2: Neither x, a or b have changed between these two statements (it only consists of y = a * c, which does not change either of these values.)

It qualifies for **common subexpression elimination**!

Common Subexpression Elimination

Consider the code below, which needs common subexpression elimination.

$$x = a + b$$

$$y = a * c$$

$$z = x$$

$$w = a * c$$

$$t = x + z$$

Step 3: Replace z = a + b with z = x.

Common Subexpression Elimination

Consider the code below, which needs common subexpression elimination.

$$x = a + b$$

$$\mathbf{y = a * c}$$

$$z = x$$

$$\mathbf{w = a * c}$$

$$t = x + z$$

Step 4: Rinse and repeat?

Common Subexpression Elimination

Consider the code below, which needs common subexpression elimination.

$$x = a + b$$

$$\mathbf{y = a * c}$$

$$z = x$$

$$\mathbf{w = y}$$

$$t = x + z$$

Step 4: Rinse and repeat?

Implementing CSE

To perform CSE within each basic block, do the following steps:

1. Initialize an empty dictionary or hash table to store expressions and their corresponding variables.
2. Iterate through the statements in the basic block.
3. For each statement, check if the expression on the right-hand side is already present in the dictionary.
    A. If the expression is not present in the dictionary, add the expression as a key and the variable on the left-hand side as the value.
    B. If the expression is present in the dictionary, replace the right-hand side of the statement with the corresponding variable from the dictionary.
4. Update the basic block with the optimized statements.

Implementing CSE: structure and search

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

// Define a dictionary-kind of structure
// to hold expressions and corresponding variables
typedef struct {
    char expression[20];
    char variable[20];
} CSETable;

// Function to search if an expression already appears in the CSE table
// Will return index i if found; and -1 otherwise.
int find_expression(CSETable *table, int table_size, const char *expression) {
    for (int i = 0; i < table_size; ++i) {
        if (strcmp(table[i].expression, expression) == 0) {
            return i;
        }
    }
    return -1;
}
```

Implementing CSE: optimization

```c
// Function to optimize a basic block using the CSE technique
void optimize_cse(const char *block[], int block_size) {
    CSETable cse_table[block_size];
    int table_size = 0;
    char lhs[20], rhs[20];

    // Iterate through the statements in the basic block
    for (int i = 0; i < block_size; ++i) {
        // Extract the left-hand side and right-hand side of the statement
        sscanf(block[i], "%[^=] = %[^\n]", lhs, rhs);
        // Check if the expression is present in the CSE table
        int index = find_expression(cse_table, table_size, rhs);
        if (index != -1) {
            // If the expression is found, print the optimized statement
            printf("%s = %s\n", lhs, cse_table[index].variable);
        } else {
            // If the expression is not found, add it to the CSE table
            strcpy(cse_table[table_size].expression, rhs);
            strcpy(cse_table[table_size].variable, lhs);
            // Print the original statement
            printf("%s\n", block[i]);
            // Increment the table size
            ++table_size;
        }
    }
}
```

Implementing CSE: optimization

```c
int main() {
    // Define a basic block as an array of strings for testing
    const char *block[] = {
        "x = a + b",
        "y = a * c",
        "z = a + b",
        "w = a * c",
        "t = x + z"
    };
    int block_size = sizeof(block) / sizeof(block[0]);
    // Optimize the basic block using the CSE technique
    optimize_cse(block, block_size);
    return 0;
}
```

In Code files/1.

```
x = a + b
y = a * c
z  = x
w  = y
t = x + z
```

Copy Propagation

Definition (Copy Propagation):

If, in the same basic block, we have two variable assignments

$$v1 = a$$

$$\ldots$$

$$v2 = v1$$

And the values of v1 or a have not changed between the declarations of v1 and v2, we can perform **copy propagation (CP)** and rewrite the v2 assignment as

$$v2 = a$$

It will help to immensely simplify our code later on.

Copy Propagation

Consider the code below, which needs copy propagation.

$$x = a + b$$

$$y = a * c$$

$$z = x$$

$$w = z$$

$$t = x + z$$

Copy Propagation

Consider the code below, which needs copy propagation.

$$x = a + b$$

$$y = a * c$$

$$\boldsymbol{z = x}$$

$$\boldsymbol{w = z}$$

$$t = x + z$$

Step 1: Identify two expressions that match the copy propagation pattern.

Copy Propagation

Consider the code below, which needs copy propagation.

$$x = a + b$$

$$y = a * c$$

$$\mathbf{z = x}$$

$$\mathbf{w = z}$$

$$t = x + z$$

Step 2: The value of the right-hand variable of the second expression (in our case, z) has not changed between both statements.

The second statement therefore qualifies for **copy propagation**.

Copy Propagation

Consider the code below, which needs copy propagation.

$$x = a + b$$

$$y = a * c$$

$$\mathbf{z = x}$$

$$\mathbf{w = x}$$

$$t = x + z$$

Step 3: Replace the right-hand side of the second expression with the right-hand side value of the first expression.

Copy Propagation

Consider the code below, which needs copy propagation.

$$x = a + b$$

$$y = a * c$$

$$\boldsymbol{z = x}$$

$$\boldsymbol{w = x}$$

$$t = x + z$$

Step 4: As before, rinse and repeat as required.

Implementing CP (almost like CSE)

To perform Copy Propagation within each basic block, do the following steps:

1. Initialize an empty dictionary or hash table to store variables and their corresponding values.
2. Iterate through the statements in the basic block.
3. For each statement, check if the right-hand side is a single variable (not an expression). If the right-hand side is a single variable, search for the variable in the dictionary.
   A. If the variable is not present in the dictionary, add the variable as a key and the left-hand side as the value.
   B. If the variable is present in the dictionary, replace the right-hand side of the statement with the corresponding value from the dictionary.
4. Update the basic block with the optimized statements.

Implementing CP (almost like CSE)

Question (for now): Should it work for partial expressions also?
Should we replace the code as shown below?
What would change in the algorithm discussed in previous slide?

x = 4		***x = 4***
y = 2	⇒	*y = 2*
w = 4 *+ y*		***w = x*** *+ y*
*z = 2*w*		*z = 2*w*

Implementing CP (almost like CSE)

Question (for now): Should it work for partial expressions also?
Should we replace the code as shown below?
What would change in the algorithm discussed in previous slide?

$\mathbf{x = 4}$		$\mathbf{x = 4}$
$y = 2$	$\Longrightarrow$	$y = 2$
$\mathbf{w = 4} + y$		$\mathbf{w = x} + y$
$z = 2*w$		$z = 2*w$

Answer: Yes! This probably should be allowed.
Leads to a reworked definition for copy propagation.

Copy Propagation - Updated

Definition (Copy Propagation - Updated):

If, in the same basic block, we have two variable assignments

$$v1 = a$$

$$\ldots$$

$$v2 = \ldots\ v1\ \ldots$$

And the values of v1 or a have not changed between the declarations of v1 and v2, we can perform **copy propagation** (**CP**) and rewrite the v2 expression as

$$v2 = \ldots\ a\ \ldots$$

It will help to immensely simplify our code later on.

Implementing CP (almost like CSE)

Question (for now): Should it work for partial expressions also?
Should we replace the code as shown below?
What would change in the algorithm discussed in previous slide?

$x = 4$		$x = 4$
$y = 2$	$\Rightarrow$	$y = 2$
$w = 4 + y$		$w = x + y$
$z = 2*w$		$z = 2*w$

(**Practice (for later):** Following the idea of the implementation of the CSE optimization, could you figure out how to implement the full CP?)

Dead code elimination

Definition (Dead Code Elimination):

In IR optimization, we say that a variable is **dead** if the value of that assignment is never called anywhere else.

For instance, in the code below, the variable **w is dead**.

$$x = 4$$

$$y = 2$$

$$z = x + y$$

$$\boldsymbol{w = z + 2}$$

$$q = z + 1$$

$$printf(q)$$

Dead code elimination

Definition (Dead Code Elimination):

In IR optimization, we say that a variable is **dead** if the value of that assignment is never called anywhere else.

Dead code elimination simply removes dead variables assignments from a given IR. **Dead code elimination** is then the process of removing parts of the code that do not affect the program's behavior.

This can include **unused variable assignments, unreachable code, or redundant operations**.

Implementing Dead Code Elimination

The dead code elimination is a two-step process

1. **Identify live variables, or liveness analysis:** Start by analysing the code in reverse order and mark **variables** as **live** if they are used in later operations.

2. **Remove dead variables:** Iterate through the statements in the code, and if a variable is assigned a value but is not marked as live, remove the assignment statement. Plain and simple

3. **Rinse and repeat**

Definition (live variables):

Variables are considered live **if their values are used in subsequent operations or if they are used as output**.

Live variables

To know if a variable will be used at some point, we iterate across the statements in a basic block in reverse order.

- Initially, some small set of values are known to be live (which ones depends on the particular program).
- When we see the statement a = b + c:
  - Just before the statement, a is not alive, since its value is about to be overwritten.
  - Just before the statement, both b and c are alive, since we are about to read their values.

*(**Question:** What if we have a = a + b?)*

Live variables, in action

{ }

x = 3

{ }

y = x + 2

{ }

z = 2

{ }

w = x + y

{ }

printf(w)

Consider a function that returns/prints the values of b and d at the end of its calculations.

- The TAC code for this function is given in italic.
- Here, {} denotes the list of live variables at any given point of the TAC code.

Live variables, in action

{ }

x = 3

{ x }

y = x + 2

{ x, y }

z = 2

{ x, y }

w = x + y

{ w }

printf(w)

Consider a function that returns/prints the values of b and d at the end of its calculations.

- The TAC code for this function is given in italic.
- Here, {} denotes the list of live variables at any given point of the TAC code.

Live variables, in action

$\{\ \}$

$x = 3$

$\{ x \}$

$y = x + 2$

$\{ x, y \}$

$\boldsymbol{z = 2}$

$\{ x, y \}$

$w = x + y$

$\{ w \}$

printf(w)

Using DCE:

- If a variable v is not live after its assignment, then it is a dead variable.
- DCE suggests to simply remove said statement.

Here z = 2; is a dead variable and should be removed.

Live variables, in action

$\{ \}$

$x = 3$

$\{ x \}$

$y = x + 2$

$\{ x, y \}$

$\{ x, y \}$

$w = x + y$

$\{ w \}$

printf(w)

Rinse and repeat the process once more!

Live variables, in action

$x = 3$

$y = x + 2$

$w = x + y$

printf(w)

Rinse and repeat the process once more!

- **Step 1:** Liveness analysis, first.
- **Step 2:** Dead code elimination, again.

(**Practice (for later):** How would you implement the DCE optimization? Is it again similar to what the CSE and CP do?)

Combining the local optimization techniques

The different optimizations we have seen so far all take care of just a small piece of the optimization.

- **Common subexpression elimination eliminates unnecessary statements.**
- **Copy propagation helps identify dead code.**
- **Dead code elimination removes statements that are no longer needed.**

To get maximum effect, we may have to apply these optimizations several times in a row.

Combining the local optimization techniques

Consider the TAC code below, which will be simplified

*b = a*a;*

*c = a*a;*

d = b + c;

e = b + b;

f = a;

printf(e)

Combining the local optimization techniques

Consider the TAC code below, which will be simplified

b = a*a;

c = a*a;

d = b + c;

e = b + b;

f = a;

printf(e)

→ Use CSE on these two statements.

Combining the local optimization techniques

Consider the TAC code below, which will be simplified

b = a*a;

c = b;

d = b + c;

e = b + b;

f = a;

printf(e)

→ Use CSE on these two statements.

Combining the local optimization techniques

Consider the TAC code below, which will be simplified

$$b = a*a;$$

$$\mathbf{c = b;}$$

$$d = b + c;$$

$$e = b + b;$$

$$f = a;$$

printf(e)

→ Use CP on these two statements.

Restricted

Combining the local optimization techniques

Consider the TAC code below, which will be simplified

*b = a*a;*

c = b;

d = b + ***b****;*

e = b + b;

f = a;

printf(e)

→ Use CP on these two statements.

Combining the local optimization techniques

Consider the TAC code below, which will be simplified

*b = a*a;*

c = b;

d = b + b;

e = b + b;

f = a;

printf(e)

→ Use CSE on these two statements.

Combining the local optimization techniques

Consider the TAC code below, which will be simplified

*b = a*a;*

c = b;

d = b + b;

e = d;

f = a;

printf(e)

→ Use CSE on these two statements.

Restricted

Combining the local optimization techniques

Consider the TAC code below, which will be simplified

*b = a*a;*

c = b;

d = b + b;

e = d;

f = a;

printf(e)

→ Use DCE on these two statements.

Combining the local optimization techniques

Consider the TAC code below, which will be simplified

*b = a*a;*

d = b + b;

e = d;

printf(e)

→ Use DCE on these two statements.

Combining the local optimization techniques

Consider the TAC code below, which will be simplified

*b = a*a;*

d = b + b;

e = d;

printf(e)

Question: How would you continue after this?

Question #2: Is there a precedence order we should respect on these optimization techniques?

More types of local optimizations

Arithmetic simplifications

- Replace hard operations with easier ones, e.g. rewrite *x = 4*a;* as *x = a << 2;* instead.

Short-circuit evaluation

- If *x = a and b;* and a computes as False in the TAC, then make x = False and treat b as a dead variable.

Constant folding

- If an operation consists of two literals (e.g. *x = 4*5;*), then replace it by its evaluated version *x = 20;* directly.

And many more possible local optimizations!

A quick word on global optimization

Definition (Interprocedural analysis):

Interprocedural analysis involves analysing the interactions between different functions in a program.

Interprocedural optimizations can include:

- **Function inlining** (replacing a function call with the actual function code),
- **Eliminating dead functions** (removing functions that are never called),
- And performing **constant propagation across function boundaries**.

A quick word on global optimization

Definition (Loop optimizations):

Loop optimizations focus on improving the performance of loops within a program. Examples include

- **Loop unrolling** (repeating the loop body multiple times to reduce the loop overhead),
- **Loop fusion** (combining multiple loops that have the same iteration space, enumerate/zip style),
- And **loop-invariant code motion** (moving code that does not change within the loop outside the loop).

A quick word on global optimization

Definition (Global dead code elimination):

We could extend this local technique to the globality of the control-flow diagram to remove unused code, variables, or functions throughout the entire program, as opposed to just within basic blocks.

Global dead code elimination can help reduce code size and improve performance by eliminating unnecessary computations.

A quick word on global optimization

Definition (Global common subexpression elimination):

As with global dead code elimination, this technique identifies and eliminates redundant computations across the entire program or function scope.

By reusing the results of previous computations, global common subexpression elimination can help reduce the overall number of computations and improve performance.

A quick word on global optimization

Consider the C code on the right.

And its TAC representation.

```c
#include <stdio.h>

int multiply(int a, int b) {
    return a * b;
}

int main() {
    int a = 5;
    int b = 3;
    int c = 7;
    int sum = 0;
    int i;

    for (i = 0; i < 10; i++) {
        sum += multiply(a, b);
        sum += c;
    }

    printf("%d\n", sum);
    return 0;
}
```

```
L0:
a = 5
b = 3
c = 7
sum = 0
i = 0
jump L2
L1:
t1 = a * b
sum = sum + t1
sum = sum + c
jump L2
L2:
i = i + 1
if i < 10 goto L1
printf sum
```

A quick word on global optimization

Consider the C code on the right.

And its TAC representation.

- **Function inlining:** replace any call of multiply(a, b), with a simple a*b.

Note: *we already replaced it (the function call produced a much longer TAC, not shown on slides!)*

```c
#include <stdio.h>

int multiply(int a, int b) {
    return a * b;
}

int main() {
    int a = 5;
    int b = 3;
    int c = 7;
    int sum = 0;
    int i;

    for (i = 0; i < 10; i++) {
        sum += multiply(a, b);
        sum += c;
    }

    printf("%d\n", sum);
    return 0;
}
```

```
L0:
a = 5
b = 3
c = 7
sum = 0
i = 0
jump L2
L1:
t1 = a * b
sum = sum + t1
sum = sum + c
jump L2
L2:
i = i + 1
if i < 10 goto L1
printf sum
```

A quick word on global optimization

Consider the C code on the right.

And its TAC representation.

- **Loop optimization:** The computation a * b does not change within the loop, so it can be moved outside the loop to reduce redundant computations.
- Move operation to L0.

```c
#include <stdio.h>

int multiply(int a, int b) {
    return a * b;
}

int main() {
    int a = 5;
    int b = 3;
    int c = 7;
    int sum = 0;
    int i;

    for (i = 0; i < 10; i++) {
        sum += multiply(a, b);
        sum += c;
    }

    printf("%d\n", sum);
    return 0;
}
```

```
L0:
a = 5
b = 3
c = 7
sum = 0
i = 0
jump L2
L1:
t1 = a * b
sum = sum + t1
sum = sum + c
jump L2
L2:
i = i + 1
if i < 10 goto L1
printf sum
```

A quick word on global optimization

Consider the C code on the right.

And its TAC representation.

- **Global dead code and common subexpression elimination?**
- **Constant folding on some operations (e.g. a*b with a and b being constants literals)?**
- **More stuff?**

```c
#include <stdio.h>

int multiply(int a, int b) {
    return a * b;
}

int main() {
    int a = 5;
    int b = 3;
    int c = 7;
    int sum = 0;
    int i;

    for (i = 0; i < 10; i++) {
        sum += multiply(a, b);
        sum += c;
    }

    printf("%d\n", sum);
    return 0;
}
```

```
L0:
a = 5
b = 3
c = 7
sum = 0
i = 0
jump L2
L1:
t1 = a * b
sum = sum + t1
sum = sum + c
jump L2
L2:
i = i + 1
if i < 10 goto L1
printf sum
```

Last lecture

Definition (Global and interprocedural optimization):

In middle-end compilers, we call a **global optimization**, any technique that attempts to **optimize and modify some code contained in a single basic block of TAC code, using information gathered by looking at the rest of the code and other basic blocks**.

Usually a bit more advanced, as it requires to analyse the entire control-flow graph of the TAC code (sometimes very challenging!).

The optimization technique is even called **multiprocedural** if it looks at multiple control-flow graphs of multiple functions.

We will briefly discuss it, but most likely <u>out-of-scope</u>.

Quiz time!

What is the main purpose of common subexpression elimination in three-address code optimization?

A. To inline function calls

B. To reduce redundant computations

C. To remove dead code

D. To simplify loop structures

Quiz time!

What is the main purpose of common subexpression elimination in three-address code optimization?

A. To inline function calls

B. To reduce redundant computations

C. To remove dead code

D. To simplify loop structures

Restricted

Quiz time!

What is the main goal of dead code elimination in three-address code optimization?

A. To remove statements that have no effect on the program's output

B. To eliminate function calls with inlining

C. To reduce the number of loop iterations

D. To simplify arithmetic expressions

Quiz time!

What is the main goal of dead code elimination in three-address code optimization?

A. **To remove statements that have no effect on the program's output**
B. To eliminate function calls with inlining
C. To reduce the number of loop iterations
D. To simplify arithmetic expressions

Quiz time!

In three-address code optimization, which technique is used to replace occurrences of a variable with its assigned value if the value does not change in the meantime?

A. Common subexpression elimination

B. Dead code elimination

C. Copy propagation

D. Constant folding

Quiz time!

In three-address code optimization, which technique is used to replace occurrences of a variable with its assigned value if the value does not change in the meantime?

A. Common subexpression elimination

B. Dead code elimination

C. **Copy propagation**

D. Constant folding

Quiz time!

Which of the following optimization techniques can be applied both locally and globally?

A. Dead code elimination

B. Constant folding

C. Loop-invariant code motion

D. Both A and B

Quiz time!

Which of the following optimization techniques can be applied both locally and globally?

A. Dead code elimination

B. Constant folding

C. Loop-invariant code motion

D. Both A and B

Restricted

Implementing CP (almost like CSE)

Question (for now): Should it work for partial expressions also?
Should we replace the code as shown below?
What would change in the algorithm discussed in previous slide?

$x = 4$	→	$x = 4$
$y = 2$		$y = 2$
$w = 4 + y$		$w = x + y$
$z = 2*w$		$z = 2*w$

Practice (for later): Following the idea of the implementation of the CSE optimization, could you figure out how to implement the full CP?

Live variables, in action

a = b;

d = a + b;

d = a;

printf(b, d)

Rinse and repeat the process once more!

- **Step 1:** Liveness analysis, first.
- **Step 2:** Dead code elimination, again.

Practice (for later): How would you implement the DCE optimization? Is it again similar to what the CSE and CP do?

Combining the local optimization techniques

The different optimizations we have seen so far all take care of just a small piece of the optimization.

- **Common subexpression elimination eliminates unnecessary statements.**
- **Copy propagation helps identify dead code.**
- **Dead code elimination removes statements that are no longer needed.**

Practice: how would you implement these operations to get maximum effect? We may have to apply these them several times in a row.