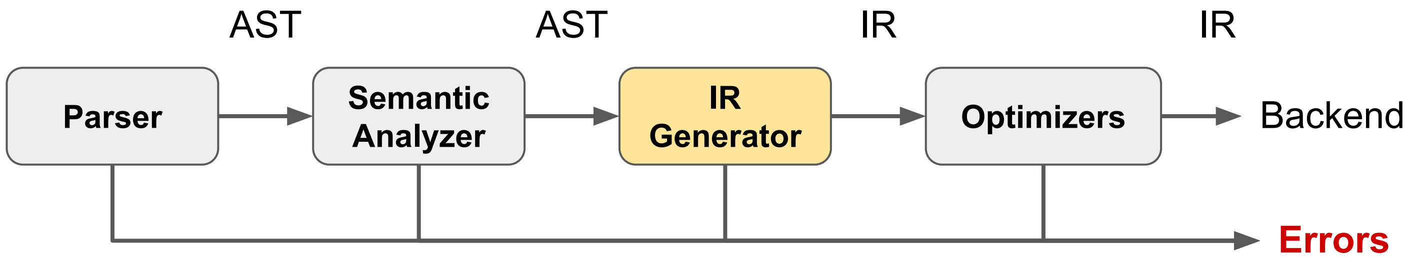
Compiling Techniques

Lecture 12: Towards SSA Form

From the Frontend towards the Backend

So far we have focused on checking the input program for *syntactic* and *semantic* errors



Now we start looking towards generating (efficient) code that is executable on hardware

Program representation for the Frontend: AST

For the analysis of the Frontend, we worked with Abstract Syntax Trees

ASTs are a natural representation of the structured program text

They are easy to work with by writing passes as recursive functions.

But they are inconvenient for *control-flow* and *data-flow* analysis, which are the basis for many compiler optimizations.

Overview: Control-flow and Data-flow Analysis

Control-flow / data-flow analysis aim to understand the program’s behaviour without executing it by analysing the possible different branches a program can take and where variables are accessed.

Crucial for these analyses are the two data structures:

- a ***def-use** chain* provides, for a single variable, the set of all its uses.
- a ***use-def** chain* provides, for a use of a variable, the set of its definitions.

Here, *definitions* means writing to a variable, and *uses* means reading from it.

Def-use and use-def chains in the AST

Example Program

```
x = 1
y = x + 1
x = 2
z = x + 1
```

Example AST in xDSL

```
assign() {
  id_expr() ["id" = "x"] } {
  literal() ["value" = 1 : !i32]  }
assign() {
  id_expr() ["id" = "y"] } {
  binary_expr() ["op" = "+"] {
    id_expr() ["id" = "x"] } {
    literal() ["value" = 1 : !i32] } }
assign() {
  id_expr() ["id" = "x"] } {
  literal() ["value" = 2 : !i32] }
assign() {
  id_expr() ["id" = "z"] } {
  binary_expr() ["op" = "+"] {
    id_expr() ["id" = "x"] } {
      literal() ["value" = 1 : !i32] } }
```

Def-use and use-def chains in the AST

Example Program

```
x = 1
y = x + 1
x = 2
z = x + 1
```

?

Example AST in xDSL

```
assign() {
  id_expr() ["id" = "x"] } {
  literal() ["value" = 1 : !i32]  }
assign() {
  id_expr() ["id" = "y"] } {
  binary_expr() ["op" = "+"] {
    id_expr() ["id" = "x"] } {
    literal() ["value" = 1 : !i32] } }
assign() {
  id_expr() ["id" = "x"] } {
  literal() ["value" = 2 : !i32] }
assign() {
  id_expr() ["id" = "z"] } {
  binary_expr() ["op" = "+"] {
    id_expr() ["id" = "x"] } {
      literal() ["value" = 1 : !i32] } }
```

Idea: Change Representation that makes def-use chains explicit

As a first step, we translate the nested AST representation into a graph representation:

AST

```
assign() {
  id_expr() ["id" = "x"] } {
  literal() ["value" = 1 : !i32]  }
assign() {
  id_expr() ["id" = "y"] } {
  binary_expr() ["op" = "+"] {
    id_expr() ["id" = "x"] } {
    literal() ["value" = 1 : !i32] } }
assign() {
  id_expr() ["id" = "x"] } {
  literal() ["value" = 2 : !i32] }
assign() {
  id_expr() ["id" = "z"] } {
  binary_expr() ["op" = "+"] {
    id_expr() ["id" = "x"] } {
       literal() ["value" = 1 : !i32] } }
```

$\longrightarrow$

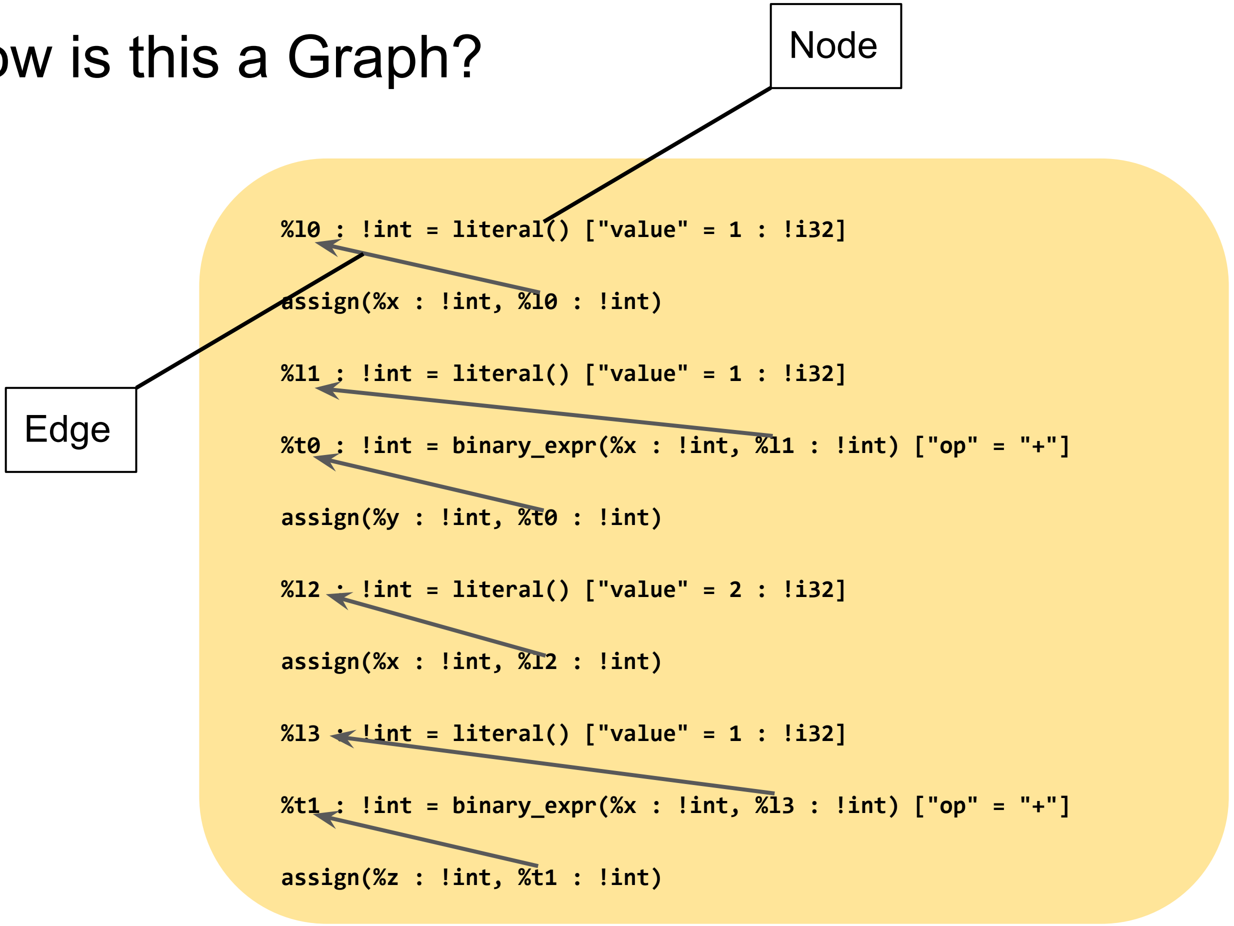
Graph-based IR

```
%l0 : !int = literal() ["value" = 1 : !i32]
assign(%x : !int, %l0 : !int)
%l1 : !int = literal() ["value" = 1 : !i32]
%t0 : !int = binary_expr(%x : !int, %l1 : !int) ["op" = "+"]
assign(%y : !int, %t0 : !int)
%l2 : !int = literal() ["value" = 2 : !i32]
assign(%x : !int, %l2 : !int)
%l3 : !int = literal() ["value" = 1 : !i32]
%t1 : !int = binary_expr(%x : !int, %l3 : !int) ["op" = "+"]
assign(%z : !int, %t1 : !int)
```

How is this a Graph?

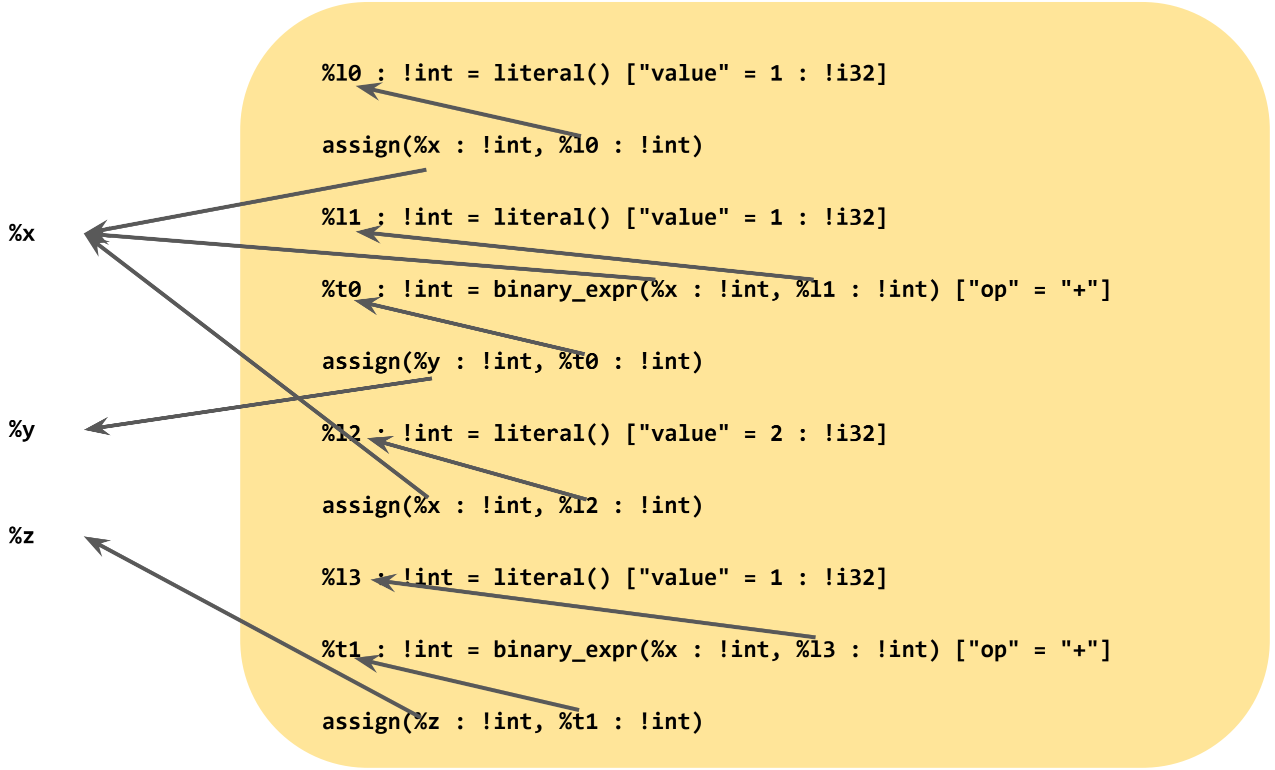
```
%l0 : !int = literal() ["value" = 1 : !i32]

assign(%x : !int, %l0 : !int)

%l1 : !int = literal() ["value" = 1 : !i32]

%t0 : !int = binary_expr(%x : !int, %l1 : !int) ["op" = "+"]

assign(%y : !int, %t0 : !int)

%l2 : !int = literal() ["value" = 2 : !i32]

assign(%x : !int, %l2 : !int)

%l3 : !int = literal() ["value" = 1 : !i32]

%t1 : !int = binary_expr(%x : !int, %l3 : !int) ["op" = "+"]

assign(%z : !int, %t1 : !int)
```

How is this a Graph?



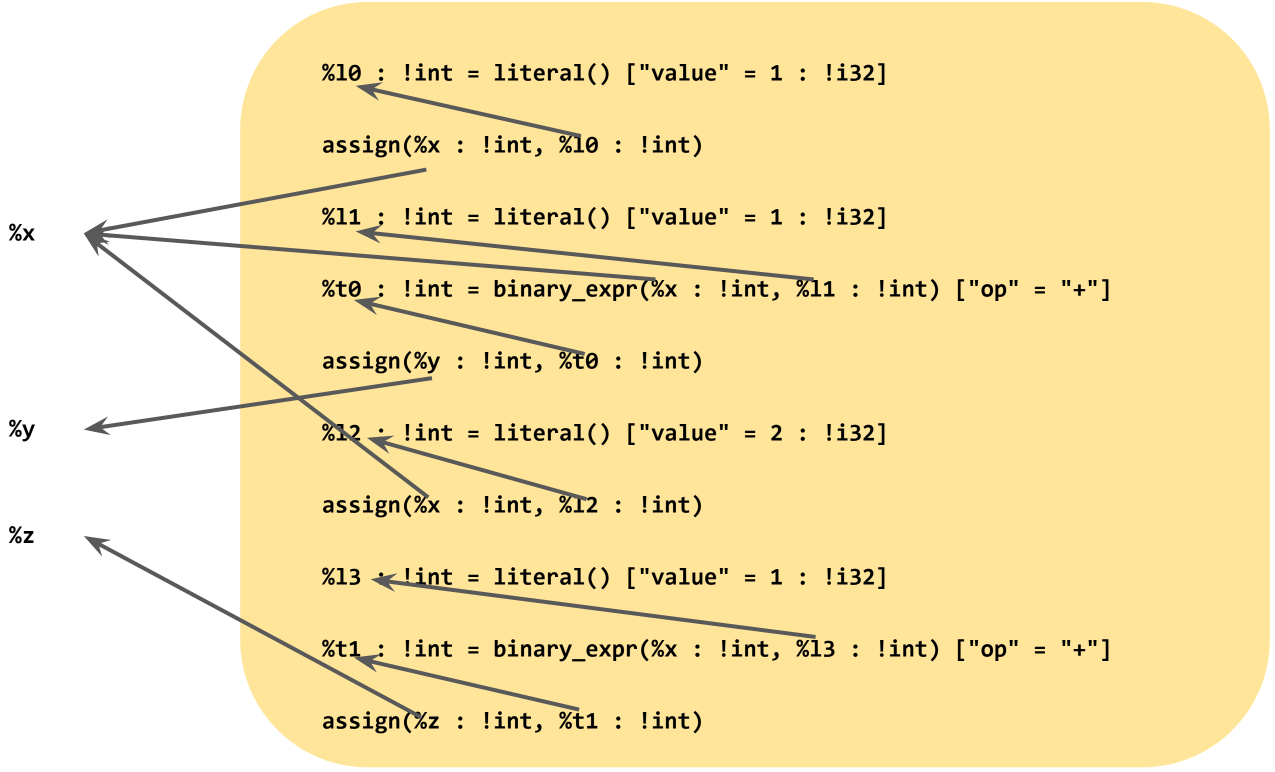
```
%l0 : !int = literal() ["value" = 1 : !i32]

assign(%x : !int, %l0 : !int)

%l1 : !int = literal() ["value" = 1 : !i32]

%t0 : !int = binary_expr(%x : !int, %l1 : !int) ["op" = "+"]

assign(%y : !int, %t0 : !int)

%l2 : !int = literal() ["value" = 2 : !i32]

assign(%x : !int, %l2 : !int)

%l3 : !int = literal() ["value" = 1 : !i32]

%t1 : !int = binary_expr(%x : !int, %l3 : !int) ["op" = "+"]

assign(%z : !int, %t1 : !int)
```

How is this a Graph?



```
%l0 : !int = literal() ["value" = 1 : !i32]

assign(%x : !int, %l0 : !int)

%l1 : !int = literal() ["value" = 1 : !i32]

%t0 : !int = binary_expr(%x : !int, %l1 : !int) ["op" = "+"]

assign(%y : !int, %t0 : !int)

%l2 : !int = literal() ["value" = 2 : !i32]

assign(%x : !int, %l2 : !int)

%l3 : !int = literal() ["value" = 1 : !i32]

%t1 : !int = binary_expr(%x : !int, %l3 : !int) ["op" = "+"]

assign(%z : !int, %t1 : !int)
```

How is this a Graph?



```
%l0 : !int = literal() ["value" = 1 : !i32]

assign(%x : !int, %l0 : !int)

%l1 : !int = literal() ["value" = 1 : !i32]

%t0 : !int = binary_expr(%x : !int, %l1 : !int) ["op" = "+"]

assign(%y : !int, %t0 : !int)

%l2 : !int = literal() ["value" = 2 : !i32]

assign(%x : !int, %l2 : !int)

%l3 : !int = literal() ["value" = 1 : !i32]

%t1 : !int = binary_expr(%x : !int, %l3 : !int) ["op" = "+"]

assign(%z : !int, %t1 : !int)
```

def $\longleftarrow$ use

def-use chains
are made explicit!

Reminder: How did we represent ASTs in xDSL?

For implementing the AST we have used the xDSL framework. Reminder:

```
@irdl_op_definition
class BinOp(Operation):
    name = "BinOp"
    op = AttributeDef(StringAttr)
    lhs = SingleBlockRegionDef()
    rhs = SingleBlockRegionDef()

@irdl_op_definition
class IntLiteral(Operation):
    name = "IntLiteral"
    value = AttributeDef(IntegerAttr)
```

`Operation` is the superclass of all AST nodes

Each `Operation` has a *name*

Metadata is represented by `attributes`

A `region` represents nested structure, such as the children of a node in the AST

A *macro* generates helpful boilerplate code to make printing, testing, etc. easy

How do we represent this graph IR in xDSL?

Let us refine our understanding of xDSL concepts:

```
@irdl_op_definition
class BinOp(Operation):
    name = "BinOp"
    op:  OpAttr[StringAttr]
    lhs: Annotated[Operand, Attribute]
    rhs: Annotated[Operand, Attribute]
    result: OpResult

@irdl_op_definition
class IntLiteral(Operation):
    name = "IntLiteral"
    value: OpAttr[Attribute]
    result: OpResult
```

`Operation` is the superclass of all IR nodes

Metadata is represented by `attributes`

`Operands` represent inputs to operations, such as the left- and right-hand-side of an addition. We used `regions` with a single operation in it before.

Operations can produce `results`, allowing other operations to refer to the computed result of an operation.

ChocoPy IR in xDSL – `Regions` only for nesting

We use `Regions` to represent nesting. In the AST, everything was nested. In the IR, we use Operands for inputs, but we still use regions to represent nesting in the input program, e.g. for the `then-` and `else-`blocks of an `If`.

IR

```
%l4 = Literal() ["value" = 4 : !i32]
%l5 = Literal() ["value" = 5 : !i32]
%res = BinaryExpr(%l4, %l5) ["op" = "+"]
```

AST

```
BinaryExpr() ["op" = "+"] {
    Literal() ["value" = 4 : !i32]
} {
    Literal() ["value" = 5 : !i32] }
```

IR

```
%t = Literal() ["value" = !bool<True>]
If(%t) {
  %l4 = Literal() ["value" = 4 : !i32]
  %l8 = Literal() ["value" = 8 : !i32]
} {
  %l15 = Literal() ["value" = 15 : !i32]
  %l16 = Literal() ["value" = 16 : !i32]
}
```

AST

```
If() {
  Literal() ["value" = !bool<True>]
} {
  Literal() ["value" = 4 : !i32]
  Literal() ["value" = 8 : !i32]
} {
  Literal() ["value" = 15 : !i32]
  Literal() ["value" = 16 : !i32]
}
```

ChocoPy IR in xDSL – `Operands` and `Results`

`Operands` and **`Results`** of Operations are written with a percent sign before a name or number: ***%x*** or ***%23***

Each of these *Names* represents a value of a certain ***type***.

Operations expect their operands to be of a certain type (similar to function argument).

In xDSL all types start with an exclamation mark: **`!int`**

```
%l0 : !int = literal() ["value" = 1 : !i32]
assign(%x : !int, %l0 : !int)
%l1 : !int = literal() ["value" = 1 : !i32]
%t0 : !int = binary_expr(%x : !int, %l1 : !int) ["op" = "+"]
assign(%y : !int, %t0 : !int)
%l2 : !int = literal() ["value" = 2 : !i32]
assign(%x : !int, %l2 : !int)
%l3 : !int = literal() ["value" = 1 : !i32]
%t1 : !int = binary_expr(%x : !int, %l3 : !int) ["op" = "+"]
assign(%z : !int, %t1 : !int)
```

Optimizing the IR

Example Program

```
x: int = 0
y: int = 0
z: int = 0

x = 1
y = x + 1
x = 2
z = x + 1
```

How can we optimize the IR?

Example Program in IR

```
module() {
  %l0 : !int = literal() ["value" = 0 : !i32]
  %x : !int = var_def(%l0 : !int) ["var_name" = "x"]
  %l1 : !int = literal() ["value" = 0 : !i32]
  %y : !int = var_def(%l1 : !int) ["var_name" = "y"]
  %l4 : !int = literal() ["value" = 0 : !i32]
  %z : !int = var_def(%l4 : !int) ["var_name" = "z"]
  %l6 : !int = literal() ["value" = 1 : !i32]
  assign(%x : !int, %l6 : !int)
  %l7 : !int = literal() ["value" = 1 : !i32]
  %t0 : !int = binary_expr(%x : !int, %l7 : !int) ["op" = "+"]
  assign(%y : !int, %t0 : !int)
  %l8 : !int = literal() ["value" = 2 : !i32]
  assign(%x : !int, %l8 : !int)
  %l10 : !int = literal() ["value" = 1 : !i32]
  %t1 : !int = binary_expr(%x : !int, %l10 : !int) ["op" = "+"]
  assign(%z : !int, %t1 : !int)
}
```

Optimizing the IR

Example Program

```
x: int = 0
y: int = 0
z: int = 0

x = 1
y = x + 1
x = 2
z = x + 1
```

How can we optimize the IR?

Remove duplicated operations

Example Program in IR

```
module() {
  %l0 : !int = literal() ["value" = 0 : !i32]
  %x : !int = var_def(%l0 : !int) ["var_name" = "x"]
  %l1 : !int = literal() ["value" = 0 : !i32]
  %y : !int = var_def(%l1 : !int) ["var_name" = "y"]
  %l4 : !int = literal() ["value" = 0 : !i32]
  %z : !int = var_def(%l4 : !int) ["var_name" = "z"]
  %l6 : !int = literal() ["value" = 1 : !i32]
  assign(%x : !int, %l6 : !int)
  %l7 : !int = literal() ["value" = 1 : !i32]
  %t0 : !int = binary_expr(%x : !int, %l7 : !int) ["op" = "+"]
  assign(%y : !int, %t0 : !int)
  %l8 : !int = literal() ["value" = 2 : !i32]
  assign(%x : !int, %l8 : !int)
  %l10 : !int = literal() ["value" = 1 : !i32]
  %t1 : !int = binary_expr(%x : !int, %l10 : !int) ["op" = "+"]
  assign(%z : !int, %t1 : !int)
}
```

Removed duplicate literals

Example Program

```
x: int = 0
y: int = 0
z: int = 0

x = 1
y = x + 1
x = 2
z = x + 1
```

Can we remove more duplicates?

Example Program in IR

```
module() {
  %l0 : !int = literal() ["value" = 0 : !i32]
  %x : !int = var_def(%l0 : !int) ["var_name" = "x"]

  %y : !int = var_def(%l0 : !int) ["var_name" = "y"]

  %z : !int = var_def(%l0 : !int) ["var_name" = "z"]
  %l6 : !int = literal() ["value" = 1 : !i32]
  assign(%x : !int, %l6 : !int)

  %t0 : !int = binary_expr(%x : !int, %l6 : !int) ["op" = "+"]
  assign(%y : !int, %t0 : !int)
  %l8 : !int = literal() ["value" = 2 : !i32]
  assign(%x : !int, %l8 : !int)

  %t1 : !int = binary_expr(%x : !int, %l6 : !int) ["op" = "+"]
  assign(%z : !int, %t1 : !int)
}
```

Removed duplicate additions

Example Program

```
x: int = 0
y: int = 0
z: int = 0

x = 1
y = x + 1
x = 2
z = x + 1
```

Can we remove more duplicates?

Remove duplicate **addition**

Example Program in IR

```
module() {
  %l0 : !int = literal() ["value" = 0 : !i32]
  %x : !int = var_def(%l0 : !int) ["var_name" = "x"]

  %y : !int = var_def(%l0 : !int) ["var_name" = "y"]

  %z : !int = var_def(%l0 : !int) ["var_name" = "z"]
  %l6 : !int = literal() ["value" = 1 : !i32]
  assign(%x : !int, %l6 : !int)

  %t0 : !int = binary_expr(%x : !int, %l6 : !int) ["op" = "+"]
  assign(%y : !int, %t0 : !int)
  %l8 : !int = literal() ["value" = 2 : !i32]
  assign(%x : !int, %l8 : !int)


  assign(%z : !int, %t0 : !int)
}
```

Remove duplicate additions

Example Program

```
x: int = 0
y: int = 0
z: int = 0

x = 1
y = x + 1
x = 2
z = x + 1
```

Example Program in IR

```
module() {
  %l0 : !int = literal() ["value" = 0 : !i32]
  %x : !int = var_def(%l0 : !int) ["var_name" = "x"]

  %y : !int = var_def(%l0 : !int) ["var_name" = "y"]

  %z : !int = var_def(%l0 : !int) ["var_name" = "z"]
  %l6 : !int = literal() ["value" = 1 : !i32]
  assign(%x : !int, %l6 : !int)

  %t0 : !int = binary_expr(%x : !int, %l6 : !int) ["op" = "+"]
  assign(%y : !int, %t0 : !int)
  %l8 : !int = literal() ["value" = 2 : !i32]
  assign(%x : !int, %l8 : !int)


  assign(%z : !int, %t0 : !int)
}
```

Can we remove more duplicates?

- Remove duplicate **addition**

This is wrong! x is mutated before the second addition.

Mutations are problematic ⇒ Remove them!

As we just saw, mutating variables is problematic when optimizing our IR

Idea: disallow mutations of variables!

Variables are initialized when they are declared and can not be modified after.

Introduce new variables instead of mutating the old one.

Mutations are problematic ⇒ Remove them!

As we just saw, mutating variables is problematic when optimizing our IR

Idea: disallow mutations of variables!

Variables are initialized when they are declared and can not be modified after.

Introduce new variables instead of mutating the old one.

Program with mutation

```
x = 1
y = x + 1
x = 2
z = x + 1
```

Program without mutation

```
x1 = 1
y = x1 + 1
x2 = 2
z = x2 + 1
```

Single Static Assignment (SSA) Form

> *"A program is defined to be in* ***Single Static Assignment (SSA)*** *form if each variable is a target of exactly one assignment statement in the program text."*

An important property from this definition is *referential transparency:*

> *"An expression is called referentially transparent if it can be replaced with its corresponding value (and vice-versa) without changing the program's behaviour".*

When our program is in SSA form, we can't make the mistake we did before!