CSc 553
Principles of Compilation

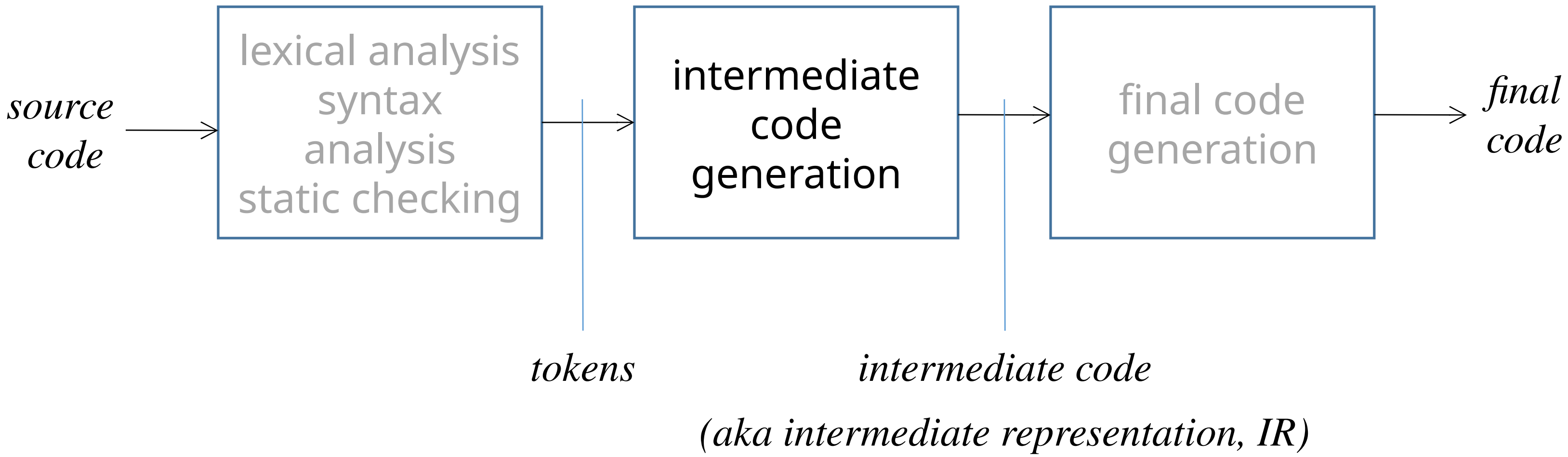
03. Intermediate Representations

Saumya Debray

The University of Arizona

Tucson, AZ 85721

Intermediate Code Generation



Why Intermediate Code?

- Closer to target language.
  - simplifies code generation.
- Machine-independent.
  - simplifies retargeting of the compiler.
  - Allows a variety of optimizations to be implemented in a machine-independent way.
- Many compilers use several different intermediate representations (IRs).

Different Kinds of IRs

- *Graphical IRs*: the program structure is represented as a graph (or tree) structure.

  Example: parse trees, syntax trees, DAGs.

- *Linear IRs*: the program is represented as a list of instructions for some virtual machine.

  Example: three-address code.

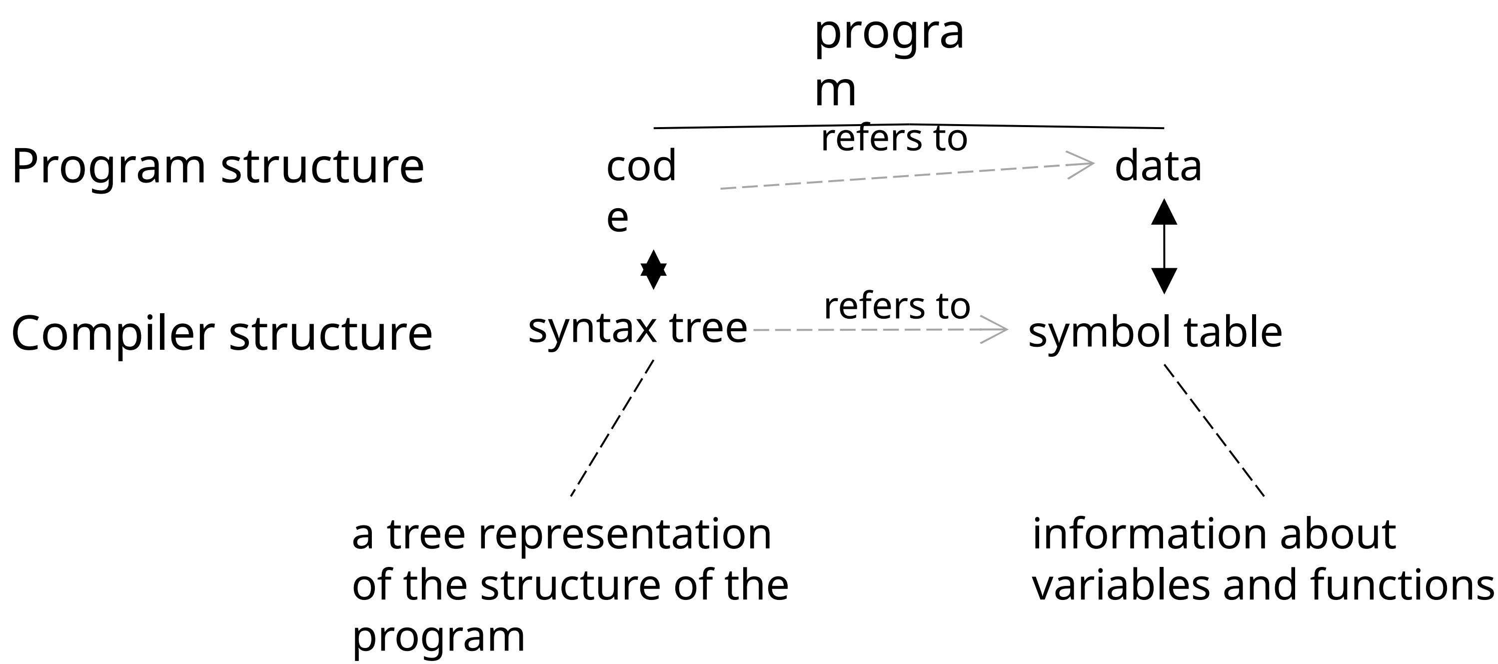
- *Hybrid IRs*: combines elements of graphical and linear IRs.

  Example: control flow graphs with 3-address code.

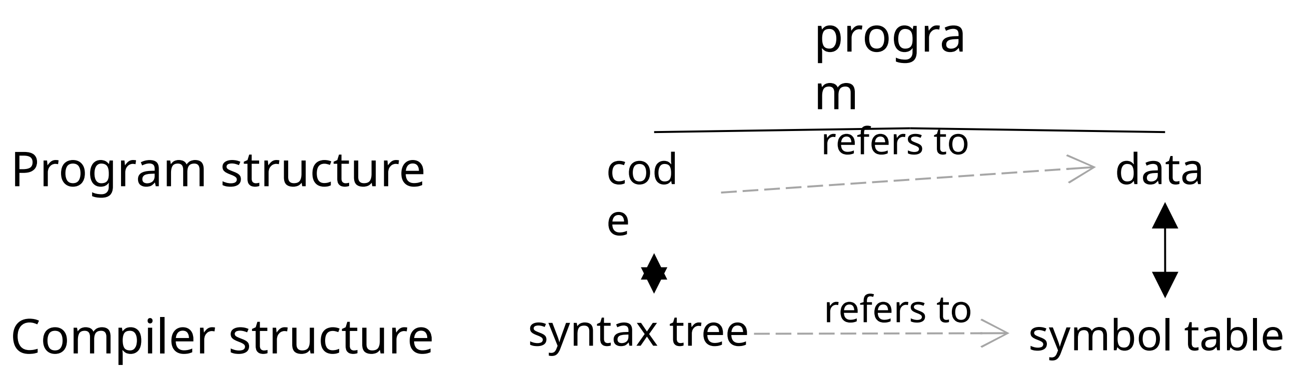
Graphical IRs: Abstract Syntax Trees

(aka "syntax trees")

Syntax Trees



Syntax Trees

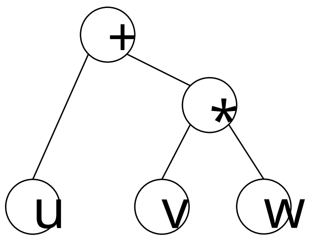


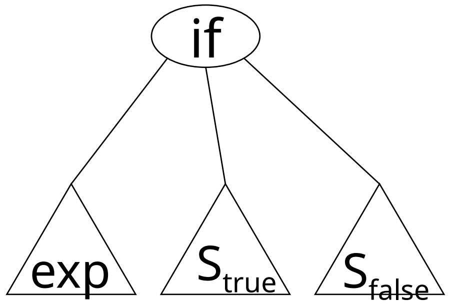
A *<u>syntax tree</u>* shows the structure of a program:

- each node represents a computation to be performed
  - leaf nodes (variable names) refer to symbol table entry
- its children represent what that computation is performed on

Syntax Trees: Structure

- Expressions:
  - leaves: identifiers or constants;
  - internal nodes are labeled with operators;
  - the children of a node are its operands.
- Statements:
  - a node's label indicates what kind of statement it is;
  - the children correspond to the components of the statement.

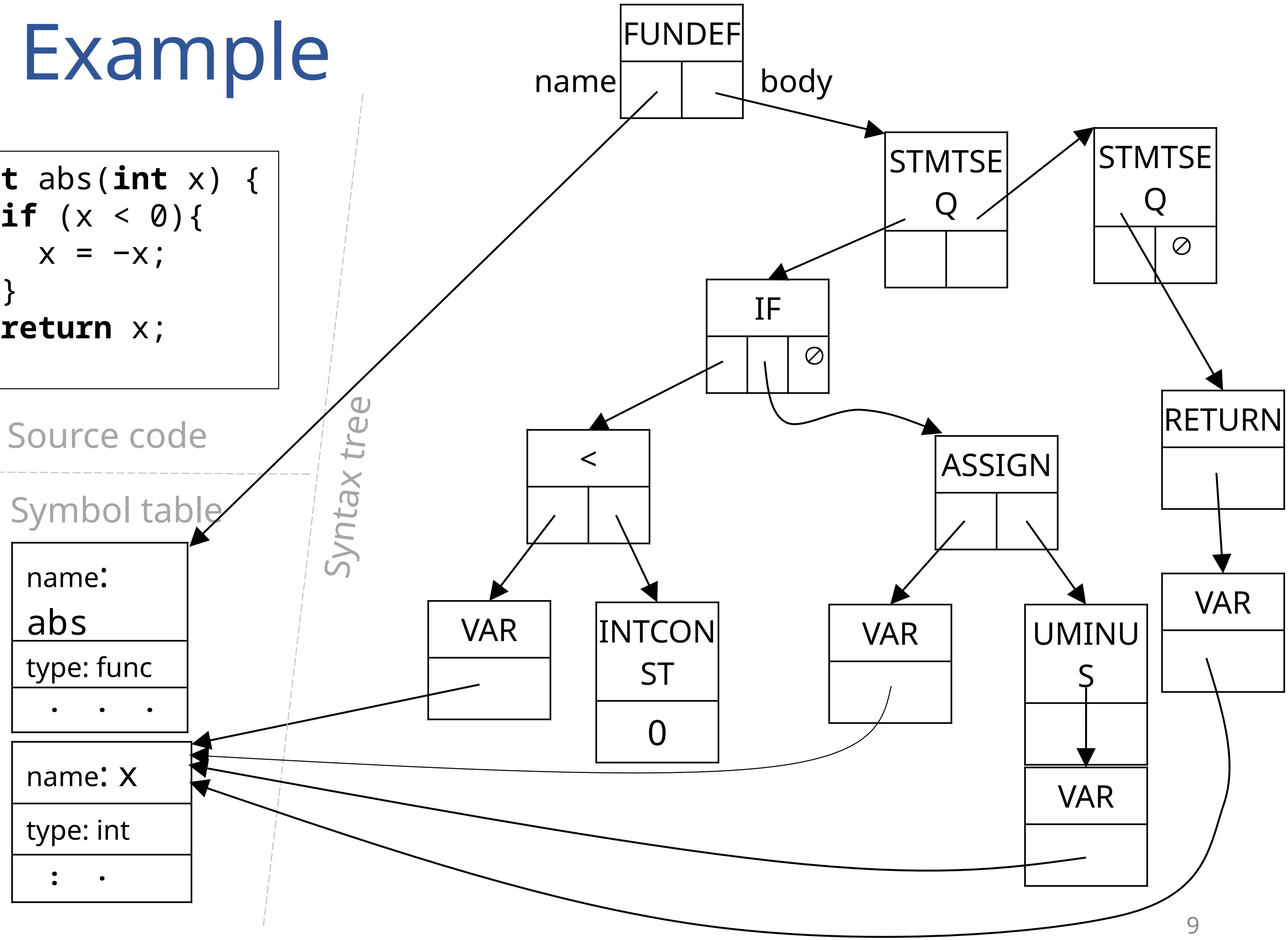




Example

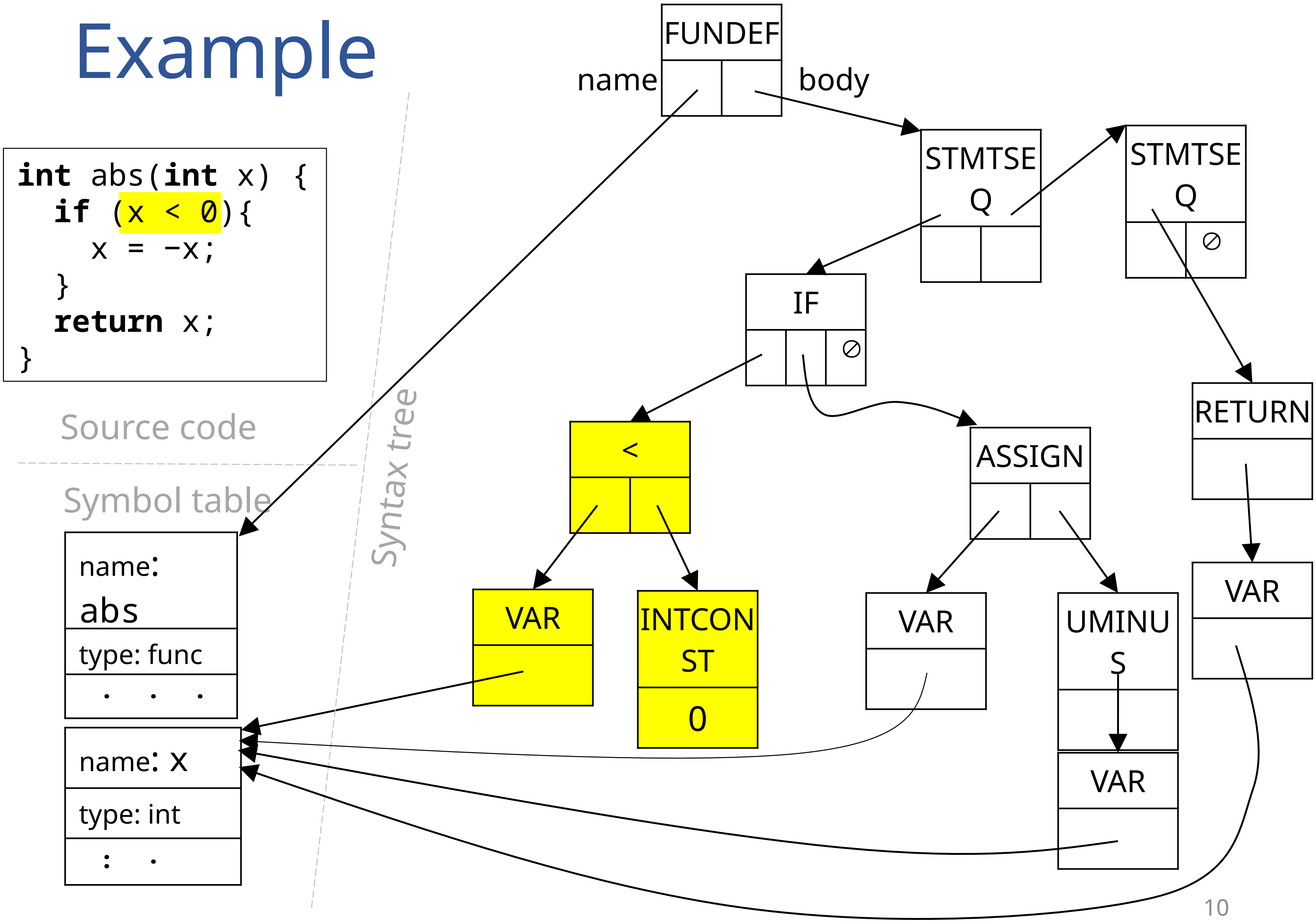
```
int abs(int x) {
  if (x < 0){
    x = -x;
  }
  return x;
}
```

Source code



Example

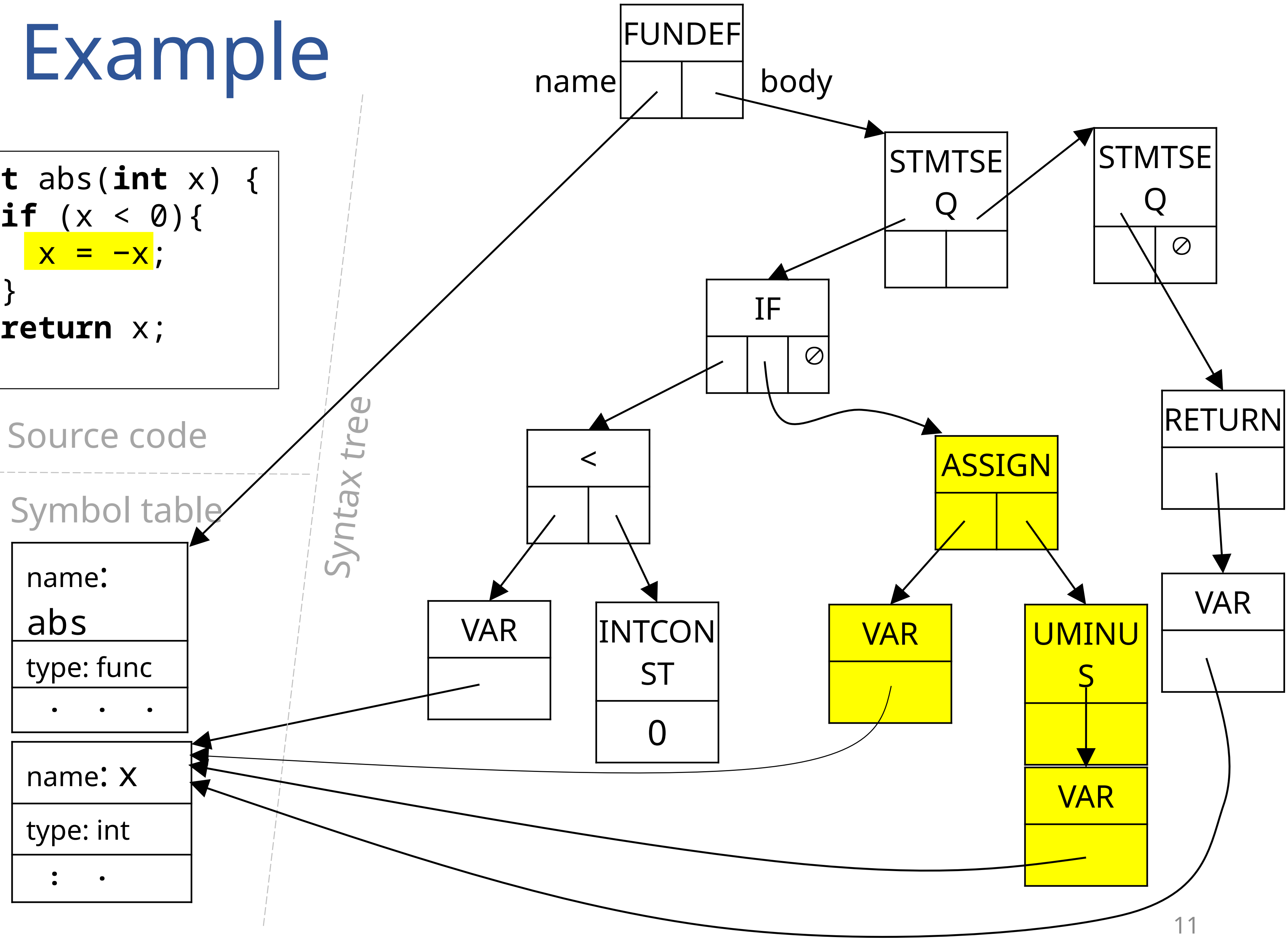
```
int abs(int x) {
  if (x < 0){
    x = -x;
  }
  return x;
}
```



Example

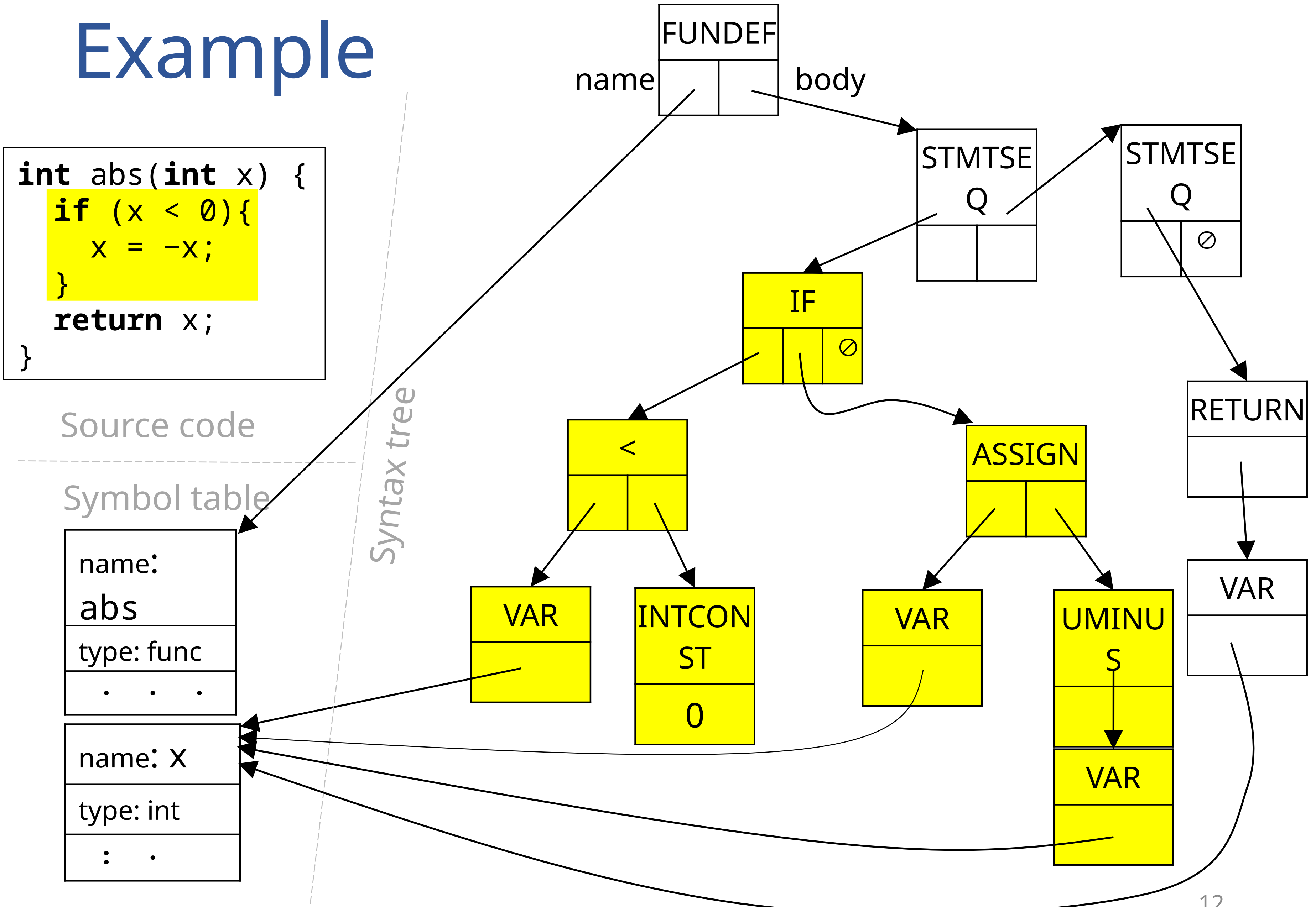
```
int abs(int x) {
  if (x < 0){
    x = -x;
  }
  return x;
}
```

Source code



Example

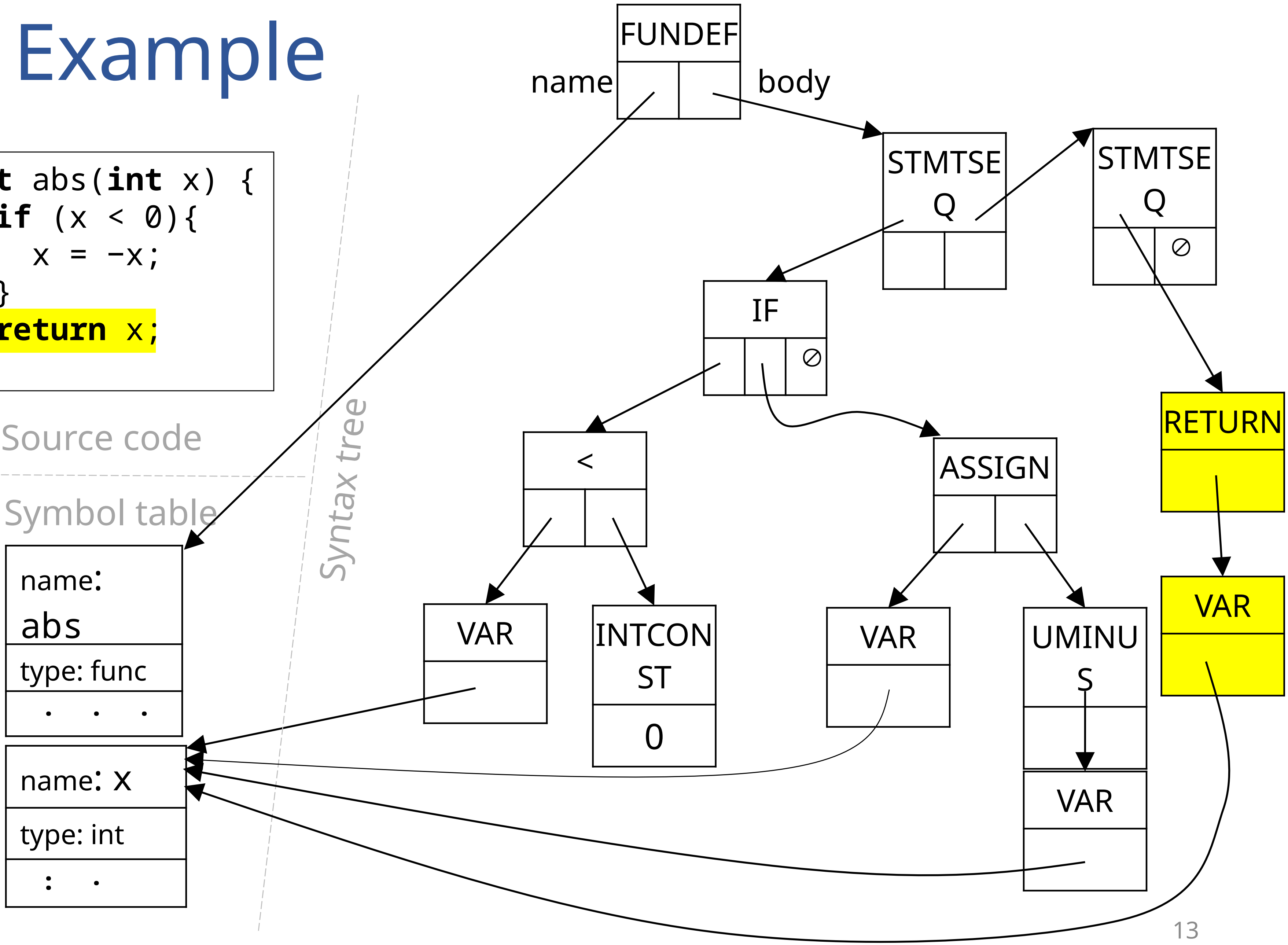
```
int abs(int x) {
  if (x < 0){
    x = -x;
  }
  return x;
}
```



Example

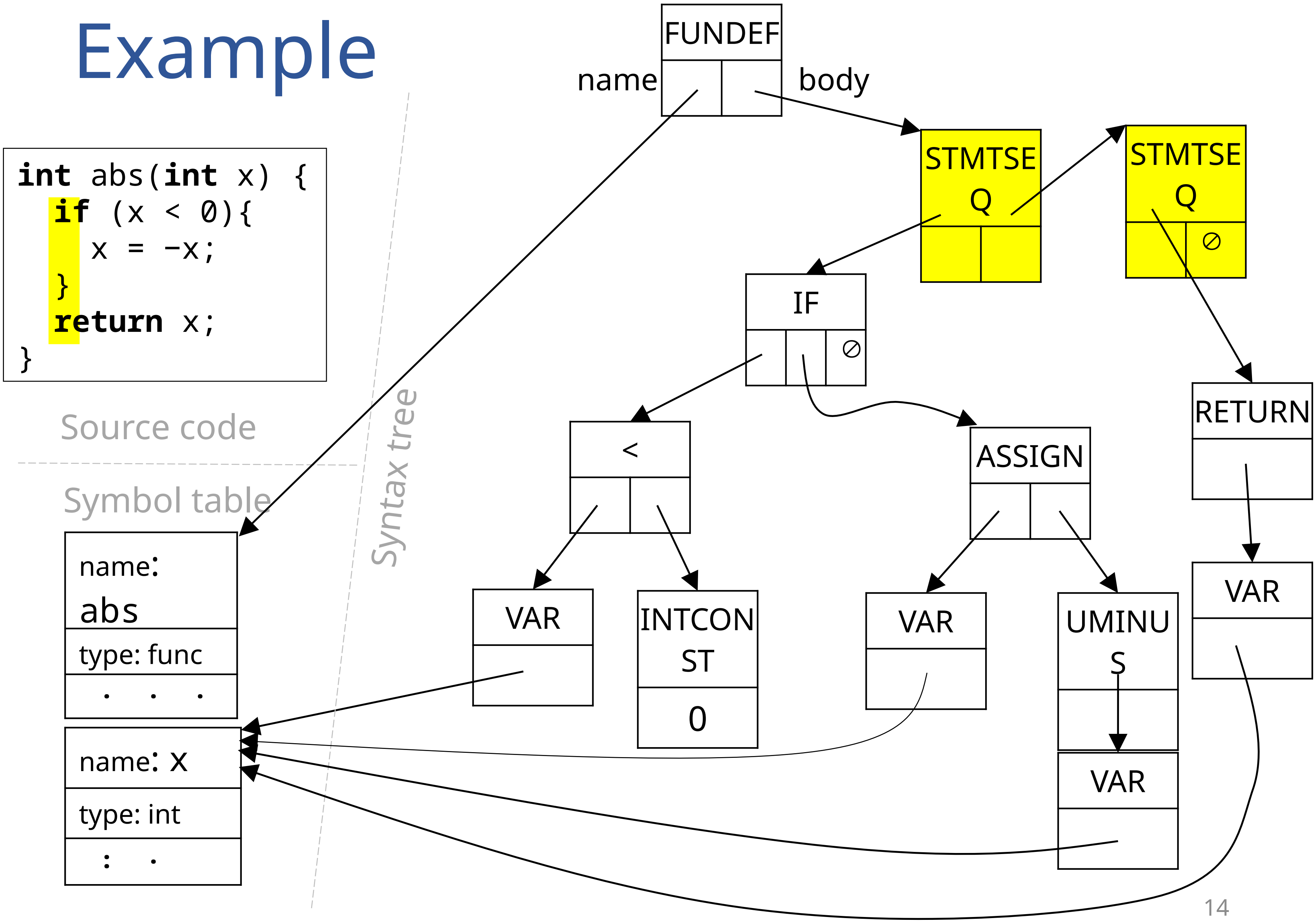
```
int abs(int x) {
  if (x < 0){
    x = -x;
  }
  return x;
}
```

Source code



Example

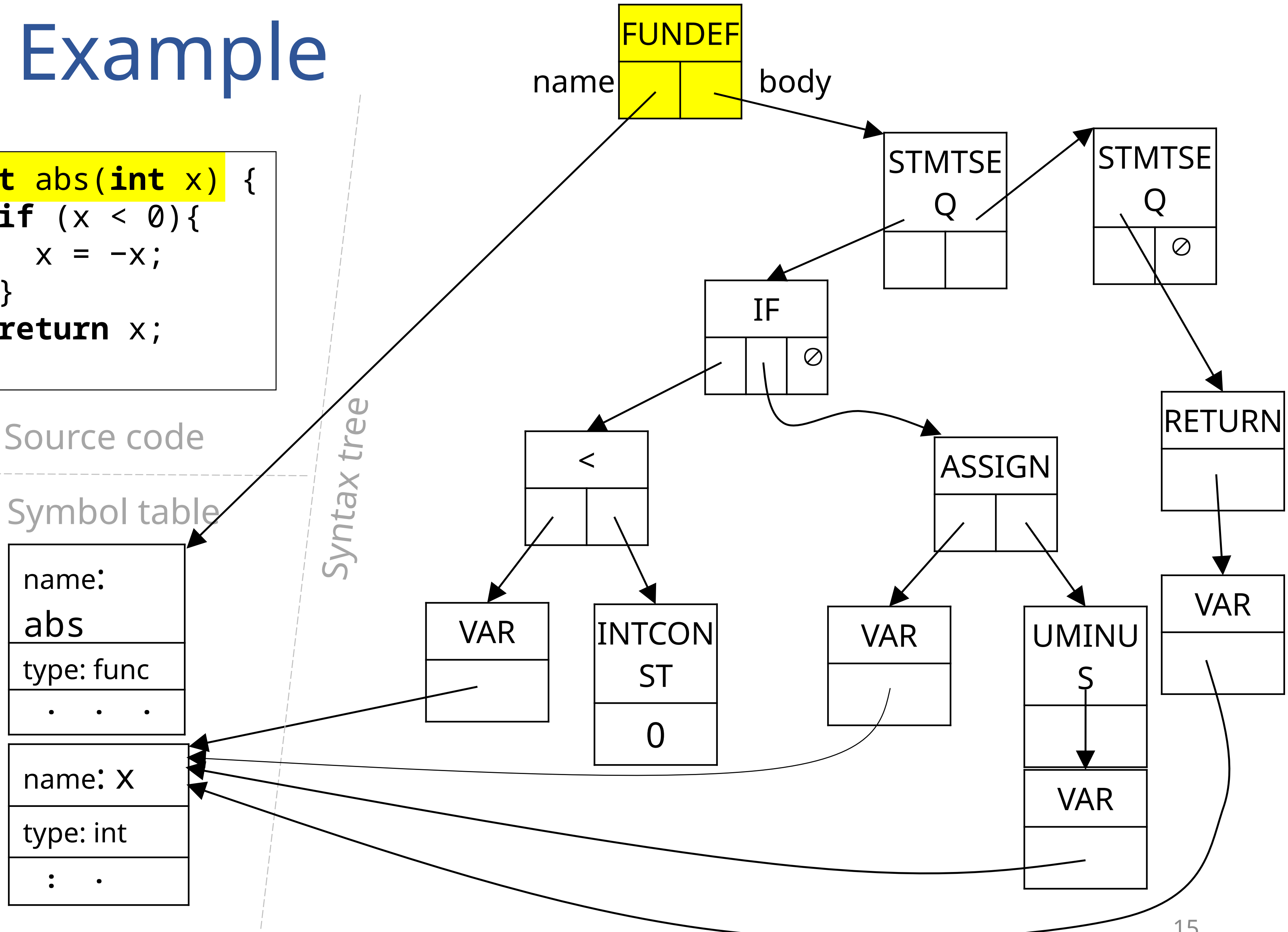
```
int abs(int x) {
  if (x < 0){
    x = -x;
  }
  return x;
}
```



Example

```
int abs(int x) {
  if (x < 0){
    x = -x;
  }
  return x;
}
```

Source code

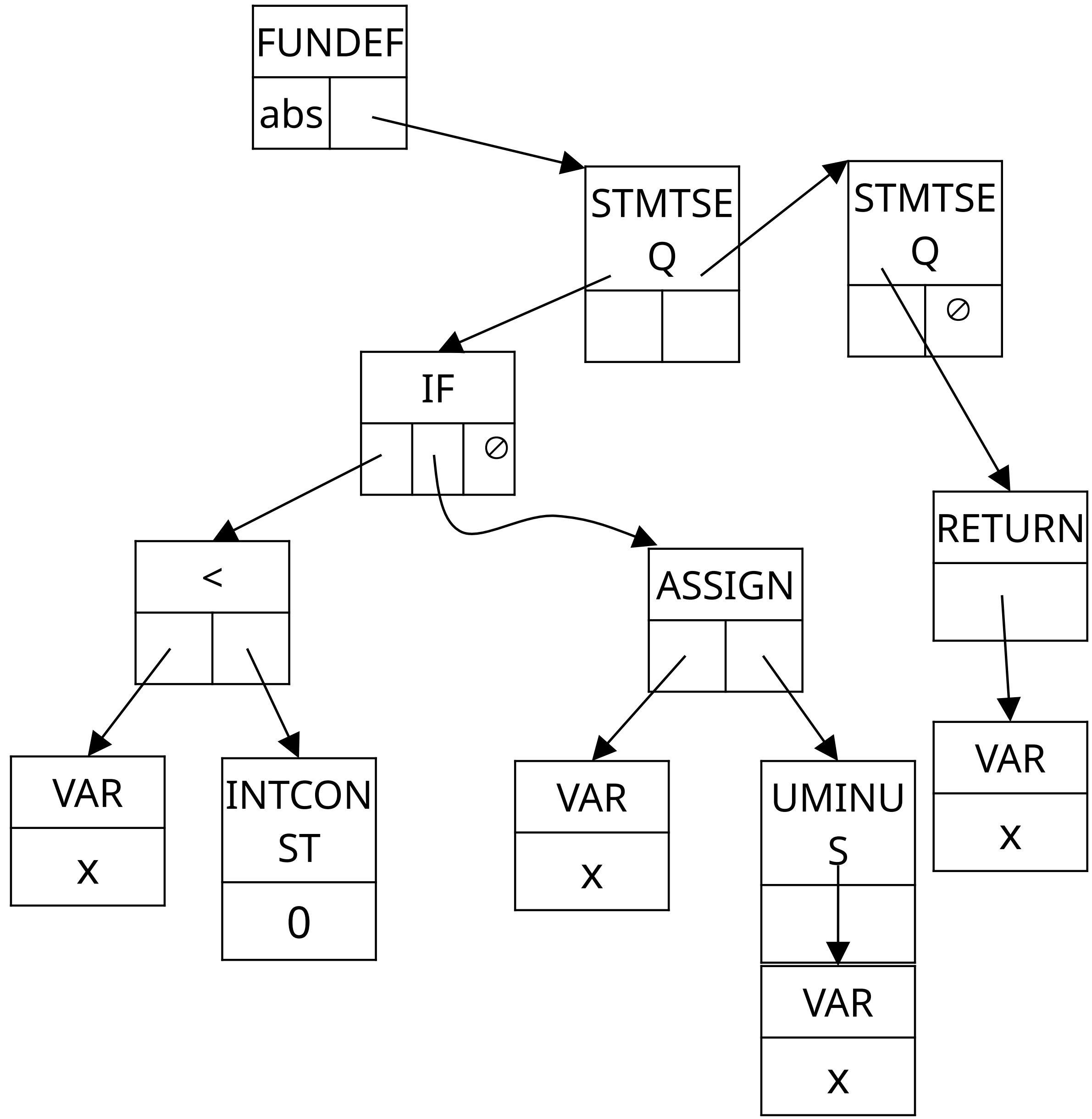


Example

```
int abs(int x) {
  if (x < 0){
    x = -x;
  }
  return x;
}
```

Source code

Syntax tree



to reduce clutter, we may omit explicit references from the syntax tree to the symbol table

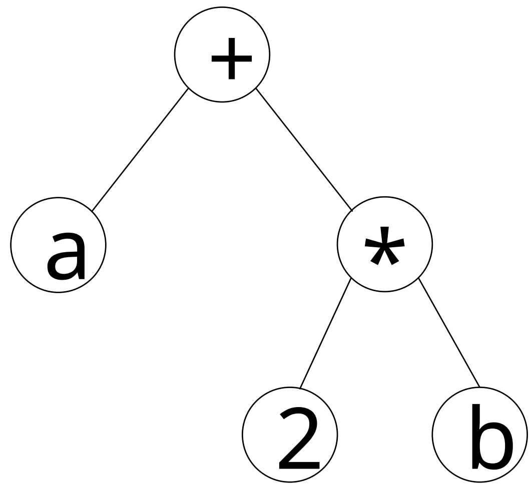
Information in syntax tree nodes

Information is added to syntax tree nodes as compilation progresses:

- parsing:
  - node type
  - children
- type checking:
  - value type
- code generation:
  - location where an expression's value is stored
  - intermediate code generated for the tree rooted at the node

EXERCISE

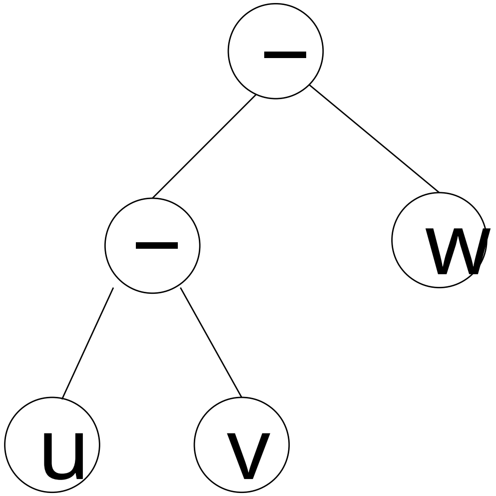
Given the syntax tree:



What source code expression(s) could this have come from?

EXERCISE

Given the syntax tree:



What source code expression(s) could this have come from?

EXERCISE

Given the syntax tree nodes: + − 1 2 3

Use exactly this set of nodes to construct a syntax tree corresponding to an expression whose value is 0.

Summary

- Syntax trees and symbol tables are used together to represent programs inside a compiler
  - syntax trees represent code structure
    - nodes represent operations, which are applied to the node's children
  - symbol tables hold information about user-defined symbols (variables, functions, etc.)
    - scope rules of C, Java, etc., require using a stack of symbol tables

Linear IRs

Linear IRs

Source language programs often use features not available in target language code. E.g.:

$$x = \underbrace{(a+b)/(c-d)*2}$$

asm doesn't support multiple operations in an instruction

- *need to save intermediate results as they are computed*

while (x > 0) { x--; y++}

asm doesn't support complex grouping of statements and arbitrary control flow

- *need a way to decompose complex control flow into something simpler*

Linear IRs

- A linear IR consists of a sequence of instructions that execute in order.
  - “machine-independent assembly code”
- Instructions may contain multiple operations, which (if present) execute in parallel.
- They often form a starting point for hybrid representations (e.g., control flow graphs).

Linear IRs 1: Three-address code

Three Address Code

- Instructions are of the form '**`x = y`** ***op*** **`z`**,' where **x**, **y**, **z** are variables, constants, or "temporaries".
- At most one operator allowed on RHS
  - no "built-up" expressions
  - instead, expressions are computed using temporaries (compiler-generated variables).
- The specific set of operators represented, and their level of abstraction, can vary widely.

Three Address Code: Example

- *Source*:

```
if ( x + y*z > x*y + z)
   a = 0;
```

- *Three Address Code*:

```
t1 = y*z
t2 = x+t1            // x + y*z
t3 = x*y
t4 = t3+z            // x*y + z
if (t2 ≤ t4) goto L
a = 0
L:
```

An Example Intermediate Instruction Set

- *Assignment*:
  - x = y *op* z (*op* binary)
  - x = *op* y (*op* unary);
  - x = y
- *Jumps*:
  - if (x *op* y) goto L (L a label);
  - goto L
- *Pointer and indexed assignments*:
  - x = y[z]
  - y[z] = x
  - x = &y
  - x = *y
  - *y = x.
- *Procedure call/return*:
  - param x, k (x is the k^{th} param)
  - retval x
  - call p
  - enter p
  - leave p
  - return
  - retrieve x
- *Type Conversion*:
  - x = cvt_A_to_B y (A, B base types) e.g.: cvt_int_to_float
- *Miscellaneous*
  - label L

EXERCISE

Source code

3-address code

x =
(a+b*2)/c

?

x = (a+b*2)/c

Three-address code:

```
tmp1 = b * 2
tmp2 = a + tmp1
tmp3 = tmp2 / c
x = tmp3
```

Example

Source code

```
int fact(int n) {
    int p = 1;
    while (n > 0) {
        p *= n;
        n –= 1;
    }
    return p;
}
```

3-address Code

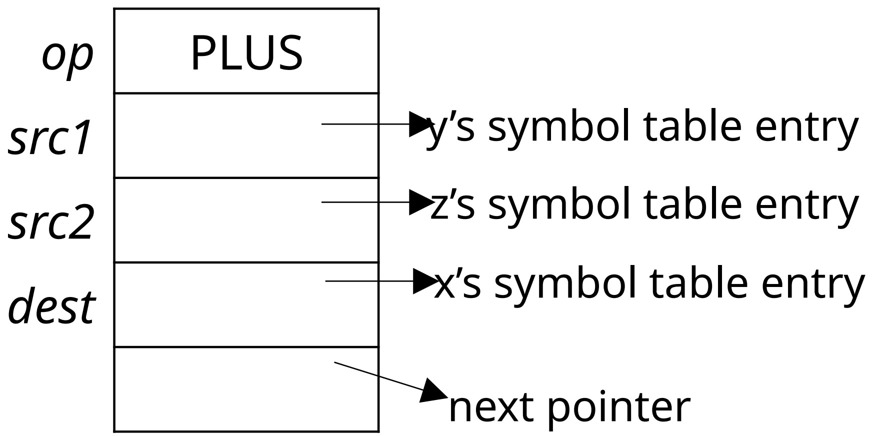
```
      enter fact
      p = 1
L0:   if n <= 0 goto L1
      tmp0 = p * n
      p = tmp0
      tmp1 = n – 1
      n = tmp1
      goto L0
L1:   leave
      return p
```

Three Address Code: Representation

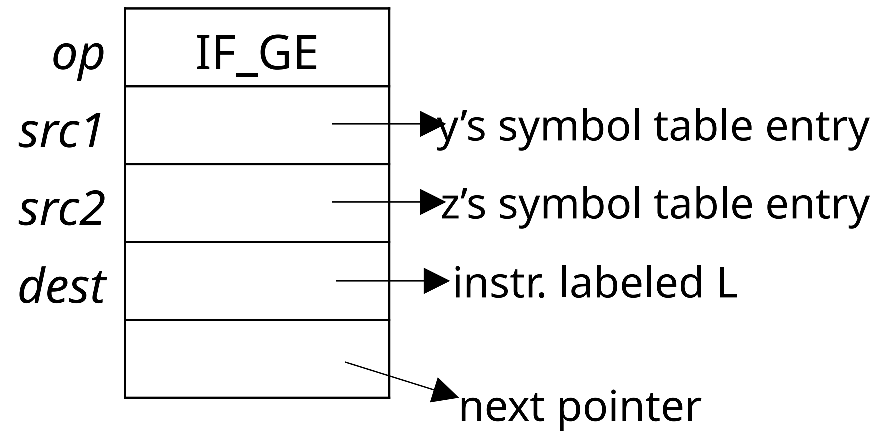
- Each instruction represented as a structure called a *quadruple* (or "*quad*"):
  - contains info about the operation, up to 3 operands.
  - for operands: use a bit to indicate whether constant or ST pointer.

E.g.:

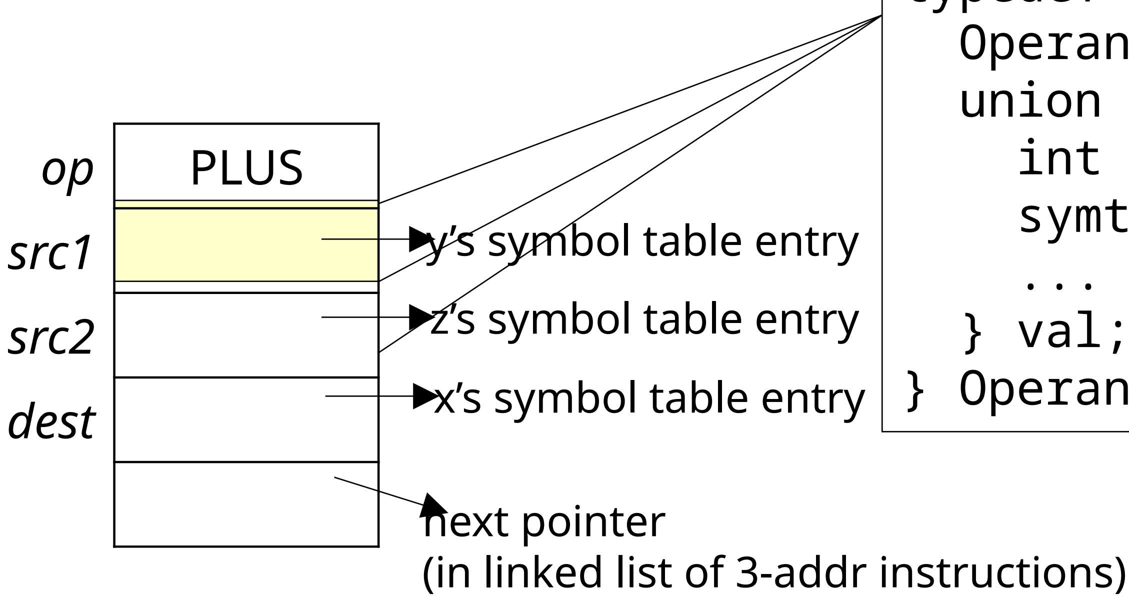
x = y + z



if (x ≥ y) goto L



Three Address Code: Representation

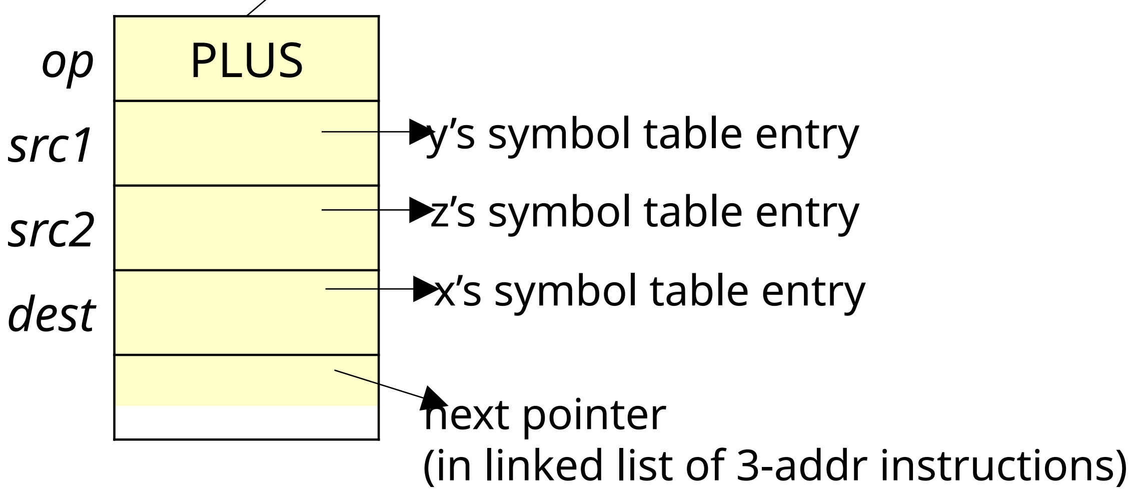



```
typedef enum {
   ... // integer constant
   ... // symbol table pointer
   ... // ...
} OperandType;

typedef struct {
  OperandType operand_type;
  union {
    int iconst;   // integer const.
    symtabnode *stptr;
    ...
  } val;
} Operand
```

Three Address Code: Representation

```
typedef struct {
    enum OpType op;      // PLUS, MINUS, etc.
    Operand src1;        // source operand 1
    Operand src2;        // source operand 2
    ...
} Quad;
```



Linear IRs 2: Stack machine code

Stack Machine Code

- Sometimes called "one-address code"
- Assumes the presence of an operand stack
  - Most operations:
    - fetch (pop) their operands from the stack
    - perform the operation
    - push the result back on the stack.

Stack-machine vs. Three-address code

Example: code for “x*y + z”

Three Address Code	Stack machine code
tmp1 = x	push x
tmp2 = y	push y
tmp3 = tmp1 * tmp2	mult
tmp4 = z	push z
tmp5 = tmp3 + tmp4	add

Stack Machine Code

- The code for an operation '*op* $x_1, \ldots, x_n$' is:

```
push xn
…
push x1
op
```

- ***Example***: JVM code for '$x = 2 \ast y - 1$':

```
iconst 1     /* push the integer constant 1 */
iload y      /* push value of integer variable y */
iconst 2
imul         /* after this, stack contains: ‹(2*y), 1〉 */
isub
istore x     /* pop stack, store to integer variable x */
```

Stack Machine Code: Features

- Compact
  - instruction operands often don’t have to be named explicitly
  - this shrinks the size of the IR.
- Necessitates new operations for manipulating the stack, e.g., “swap top two values”, “duplicate value on top.”
- Simple to generate and execute.
- Interpreted stack machine codes easy to port.

Generating Stack Machine Code

Essentially just a post-order traversal of the syntax tree:

```
void gencode(struct syntaxTreeNode ∗tnode )
{
  if ( IsLeaf( tnode ) ) {  …  }
  else {
    n = tnode→n_operands;
    for (i = n; i > 0; i--) {
      gencode( tnode→operand[i] );        /* traverse children first */
    }  /* for */
    gen_instr( opcode_table[tnode→op] );   /* code for the node */
  }  /* if [else] */
}
```

EXERCISE

Source code

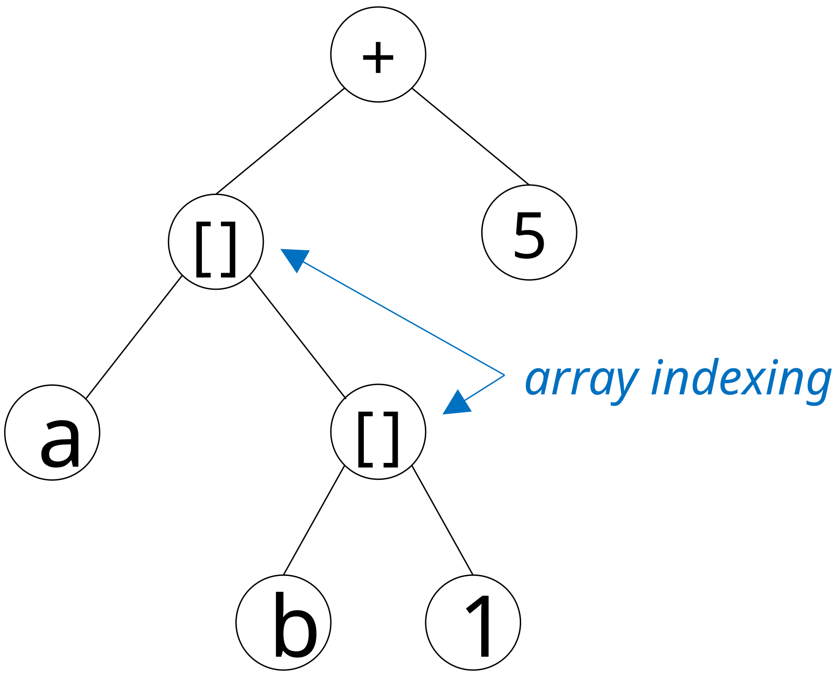
x =
(a+b*2)/c

Stack-machine code

?

EXERCISE

Syntax tree



Stack-machine code

?

EXERCISE

Stack machine code (JVM)

```
iload  a
iload  b
iadd
iload  a
iload  b
sub
div
iconst
1
add
```

Source code

?

Hybrid IRs

Hybrid IRs

- Combine features of graphical and linear IRs:
  - linear IR aspects capture a lower-level program representation;
  - graphical IR aspects make control flow behavior explicit.

- *Examples*:
  - control flow graphs
  - static single assignment form (SSA).

Hybrid IRs 1: Control Flow Graphs

Control Flow Graphs: Definition

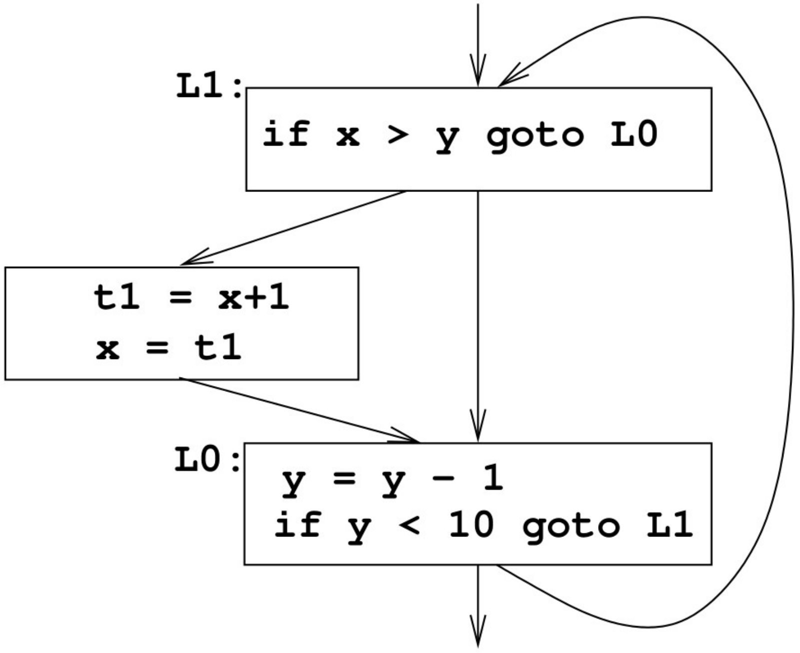
A control flow graph for a function is a directed graph $G = (V, E)$ such that:

- each $v \in V$ is a straight-line code sequence ("basic block");
- there is an edge $a \rightarrow b \in E$ iff control can go directly from a to b.

Control Flow Graphs: Example

Three-address Code	Control flow graph

```
L1: if x > y goto L0
    t1 = x+1
    x = t1
L0: y = y - 1
    if y < 10 goto L1
```



Basic Blocks

- *Definition*: A basic block *B* is a sequence of consecutive instructions such that:
    1. control enters *B* only at its beginning; and
    2. control leaves *B* only at its end (under normal execution); and

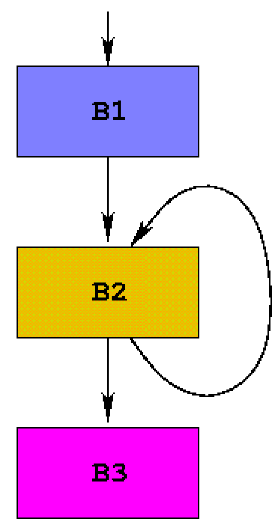
- This implies that if any instruction in a basic block B is executed, then *all* instructions in B are executed.
    - ⇒ for program analysis purposes, we can treat a basic block as a single entity.

Identifying Basic Blocks

1. Determine the set of *leaders*, i.e., the first instruction of each basic block:
   - the entry point of the function is a leader;
   - any instruction that is the target of a branch is a leader;
   - any instruction following a (conditional or unconditional) branch is a leader.
2. For each leader, its basic block consists of:
   - the leader itself;
   - all subsequent instructions up to, but not including, the next leader.

Example

```
int dotprod(int a[], int b[], int
    N) {
  int i, prod = 0;
  for (i = 1; i ≤ N; i++) {
    prod += a[i]∗b[i];
  }
  return prod;
}
```



No.	Instruction	leader?	Block No.
1	`enter dotprod`	Y	B1
2	`prod = 0`		B1
3	`i = 1`		B1
4	`t1 = 4*i`	Y	B2
5	`t2 = a[t1]`		B2
6	`t3 = 4*i`		B2
7	`t4 = b[t3]`		B2
8	`t5 = t2*t4`		B2
9	`t6 = prod+t5`		B2
10	`prod = t6`		B2
11	`t7 = i+i`		B2
12	`i = t7`		B2
13	`if i ≤ N goto 4`		B2
14	`retval prod`	Y	B3
15	`leave dotprod`		B3
16	[illegible]		[illegible]

Constructing Control flow graphs

Algorithm:

1. Identify basic blocks
2. For each block B:
   - if B ends in a branch instruction:

     add an edge to each possible control flow target
   - else:

     add an edge to the textually next basic block (i.e., the block that follows B in terms of instruction order)

Issues:

- handling function calls
- entry and exit blocks

Handling function calls

Two approaches

- treat the call instruction as ending the block
    - in this case, need to keep the next block connected to the call
- treat the call instruction as not ending a block
    - in this case, need to deal with possible side effects of the call

Hybrid IRs 2: Static Single-Assignment Form

Static Single Assignment Form: Definition

An IR is in SSA form if every assignment is to a distinct variable name

(i.e., each variable is assigned exactly once)

⇒ "single assignment"

In the program's IR, not during execution
⇒ "static"

Motivation: simplifies and enhances several analyses and optimizations

Static Single Assignment Form: Example

Three-address Code	SSA
x = a + b	$x_1 = a + b$
y = x * 2	$y_1 = x_1 * 2$
x = x – 1	$x_2 = x_1 - 1$
y = y + 1	$y_2 = y_1 + 1$
x = x * y	$x_3 = x_2 * y$
y = x / 3	$y_3 = x_3 / 3$

Static Single Assignment Form

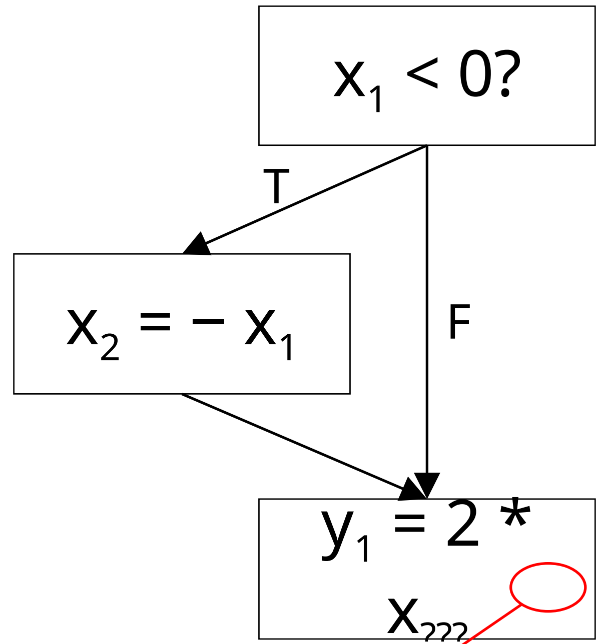
- The Static Single Assignment (SSA) form of a program makes information about variable definitions and uses explicit.
  - this can simplify program analysis
  - constructing the intermediate representation (SSA) is more work

- Most modern compilers (e.g., GCC, clang) use SSA representations.

SSA: Handling control flow merges

Code	SSA

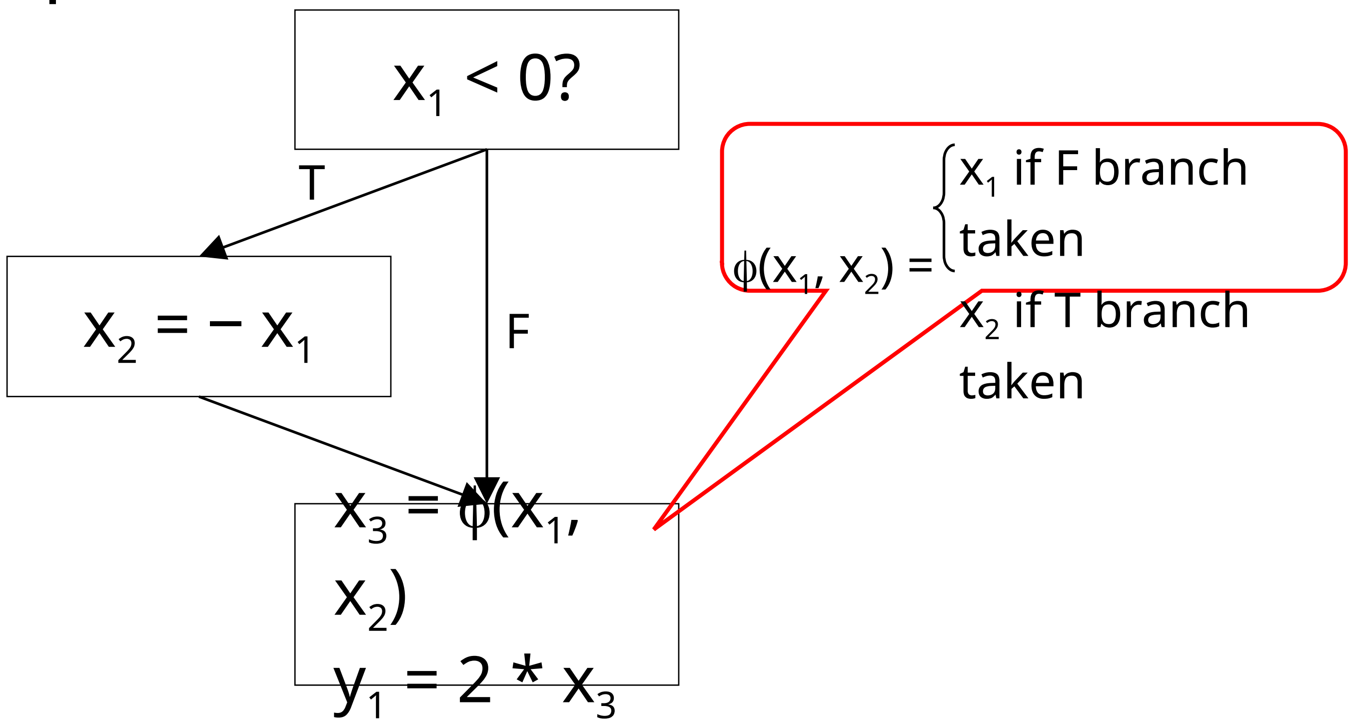
```
if (x < 0) {
    x = − x
}
y = 2 * x
```



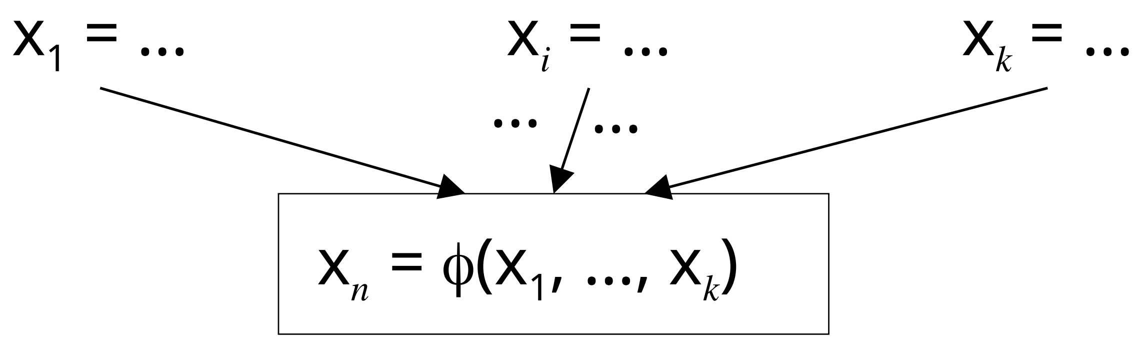
Problem: How to indicate which “version” of x to use here?

SSA: Handling control flow merges

SSA uses a notational convention called ϕ-functions to combine definitions of a variable at a merge point:



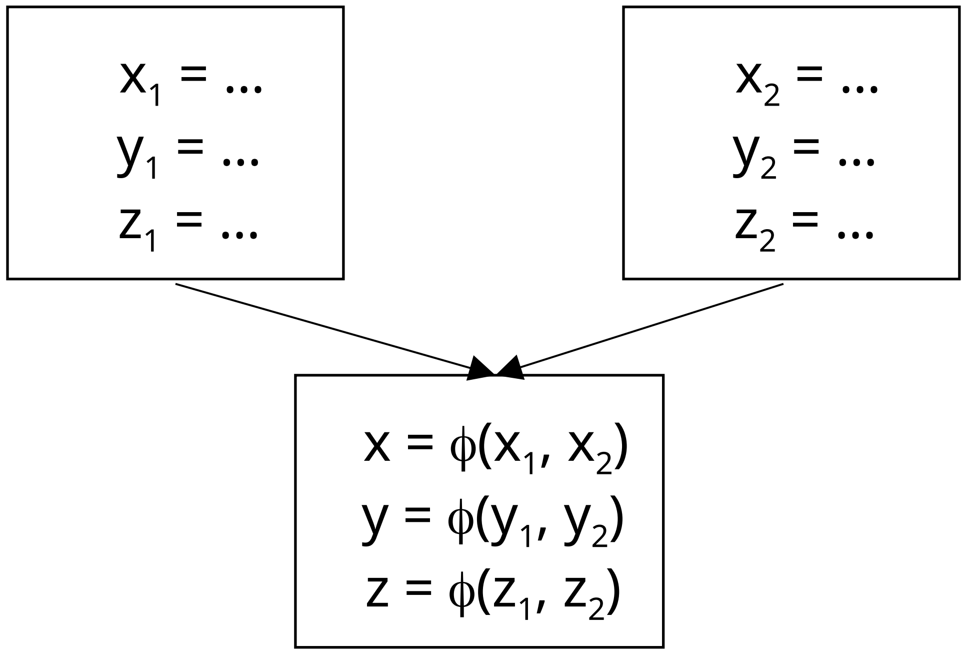
SSA: Handling control flow merges



ϕ-functions:

- placed by the compiler as necessary at merge points
- **Conceptually**: At runtime, if control reaches the ϕ-function along the branch where x_i is defined, then ϕ "selects" x_i as its value.
- **Implementation**: map $\{x_1, x_2, \ldots, x_k\}$ to the same location.

SSA Form: ϕ - Functions



On entry to a basic block, all the ϕ-functions in the block execute (conceptually) in parallel.

SSA Form: Example

Original code

```
x =
y =
while (x < 100) {
    x = x+1
    y = y+x
}
z = x*y
```

IR in SSA form

$x_0 = \ldots$
$y_0 = \ldots$

L0: $x_1 = \phi(x_0, x_2)$
$y_1 = \phi(y_0, y_2)$
if $(x_1 \geq 100)$ goto L1

$x_2 = x_1 + 1$
$y_2 = y_1 + x_2$
goto L0

L1: $z = x_1 * y_1$

SSA Form: Issues

1. Constructing a (good) SSA representation for a program

2. Mapping SSA representations to lower-level code

To be discussed later