BLEKINGE INSTITUTE OF TECHNOLOGY

BTH

Intermediate Representation

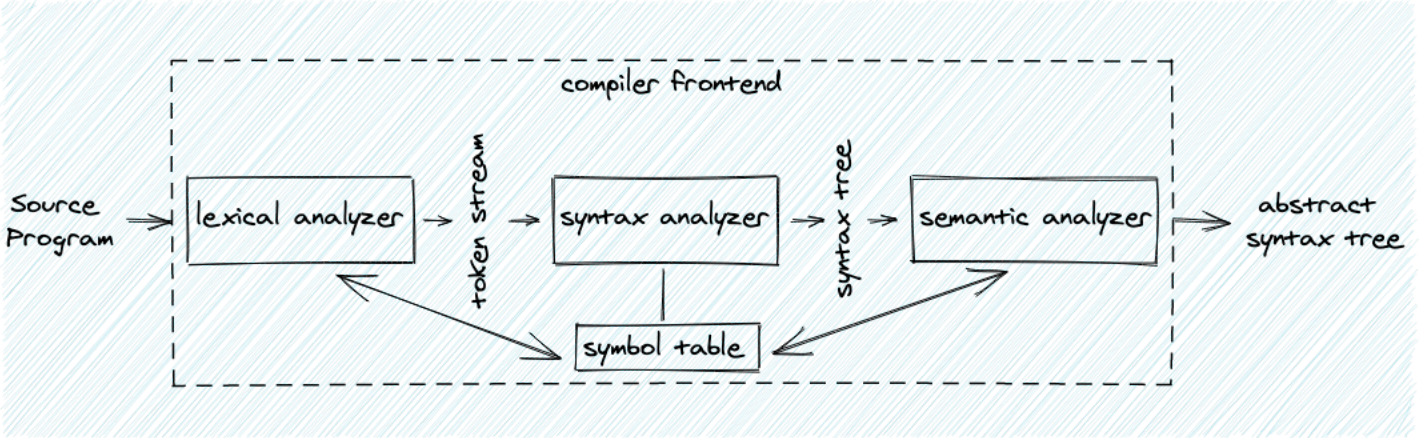
Suejb Memeti

February 2021

Outline

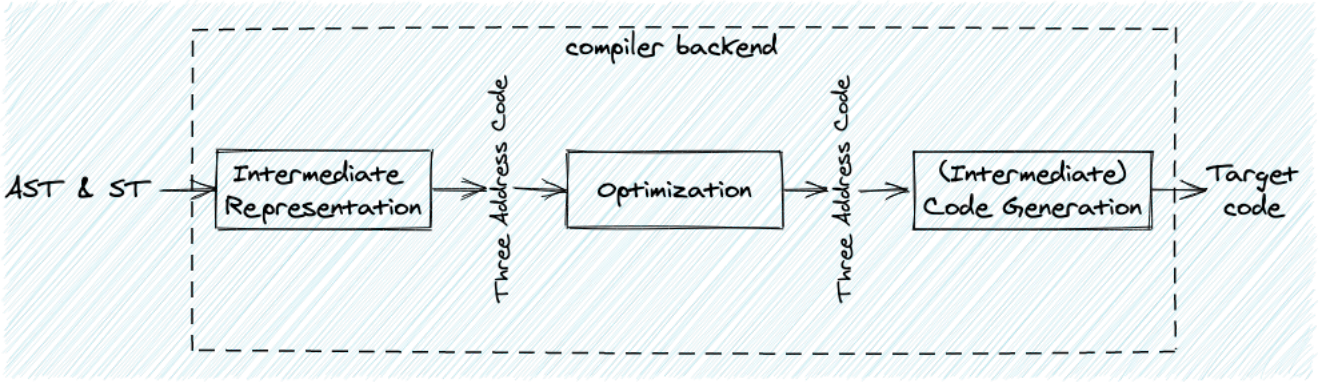
- Backend Overview
- Intermediate Representation
- Control flow graph (CFG)
  - Basic Building Blocks
  - Three Address Instructions/Code (TAC)
- Translating AST to CFG

Frontend Overview



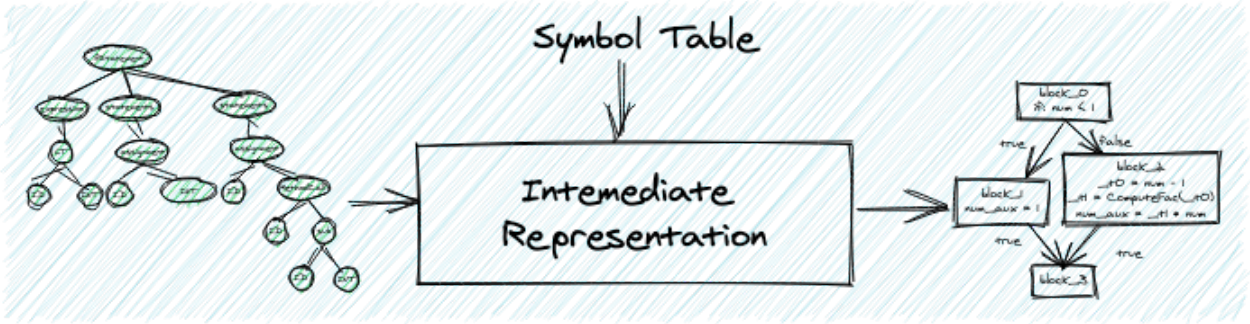
- Lexical analysis
  - Identify atomic language constructs
  - Ex: if -> IF; 2.0 -> FLOAT; x -> ID; …
- Syntax analysis
  - Check if the token stream is correct with respect to language specification
- Semantic analysis
  - Check type relations and consistency rules
  - E.g. if type(lhs) = type(rhs) in an assignment lhs=rhs; …
- Symbol table
  - Contains information about the source program. Used by all phases of the compiler.

Backend Overview



- Intermediate Representation
  - Transform the AST into a low-level intermediate representation
- Optimization
  - Perform transformations into the IR to improve performance aspects of the program
    - E.g., copy propagation, dead-code elimination, branch-prediction, loop unrolling, …
- (Intermediate) Code generation
  - Generation of the target code
    - E.g., assembly, bytecode, other high-level code (C++, Java, …)

Intermediate Representation (IR)

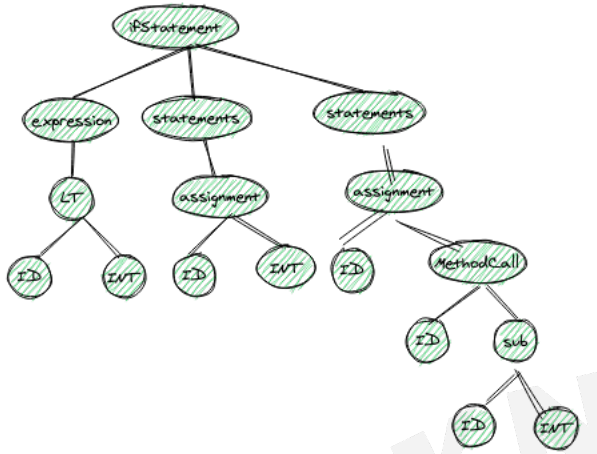


- Input: abstract syntax tree + Symbol table
- Output: Control Flow Graph (CFG) = TAC + BB

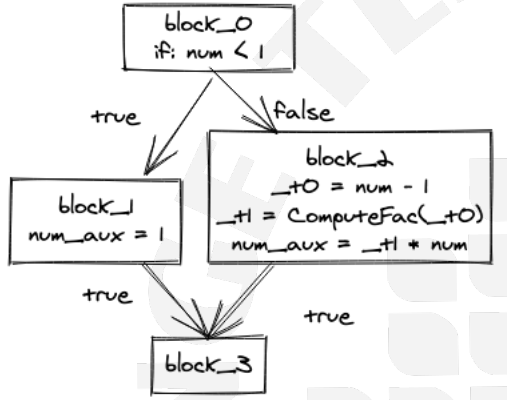
- Three Address Code (TAC): Sequence of elementary program steps
- Basic Blocks (BB): sequences of statements executed one-after-the-other

Types of IR

- Abstract Syntax Tree (AST)
  - High-level IR, hierarchical representation
  - Preserves the source code programming constructs
  - Bound to the source language



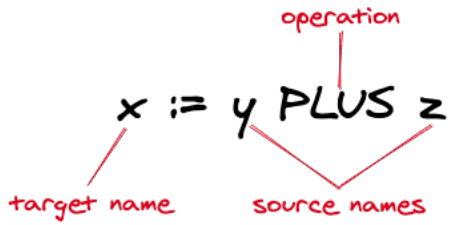
- Three Address Code (TAC)
  - Low-level IR, no hierarchy
  - Important for code optimizations
  - Independent of the source language
  - Generated by traversing the AST

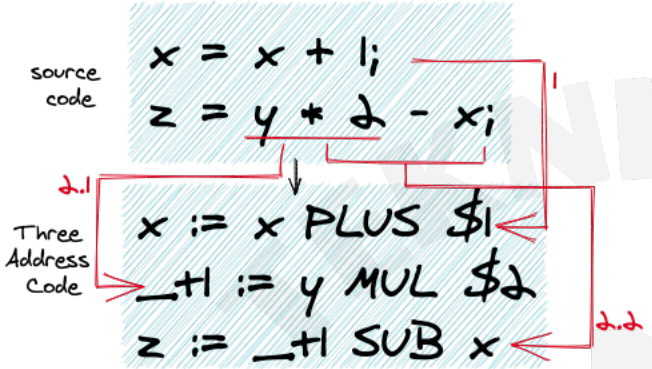


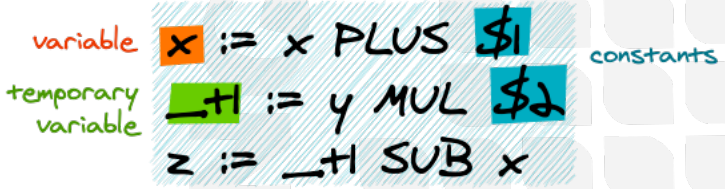
Three Address Code

- A sequence of Three Address Instructions
  - x = y **op** z
    - *x, y, z* are addresses (could be variable name, constants, or temporary variables)
    - *op* is an operator
  - At most one address on the left-hand side (target name)
  - At most one operator and two addresses (source names) on the right-hand side

- Three kinds of addresses
  - Variables – name (key) in the symbol table
    - Used by the programmer
  - Constants – special name, used as a source (read-only)
  - Temporary variables – special names
    - Used by the compiler





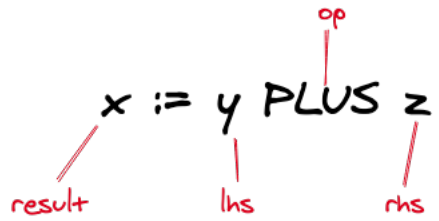


Three Address Code – examples

IR Operation	Example	Three Address Code	Notes
Expression	x = y + z	x := y op z	calculate y op z, store the result in x
Unary expression	x = !y	x := op y	calculate op y, store the result in x
Copy	x = y	x := y	copy y to x
Array Access	x = y[i]	x := y[i]	copy y[i] to x
	y[i] = x	y[i] := x	copy x to y[i]
New	x = new A;	x := new TYPE	create a new object of type A, store it in x
New Array	x = new int[N]	x := new TYPE, N	create a new object of type int[], with size N, store in x
length	x = y.length	x := length y	get the member length of y, store the results in x
Parameter	x = this.f(y)	param y	define parameter y
Method call	x = this.f(y)	x := call f, N	call function f, use N params from the stack
Return	return y	return y	return y
Unconditional Jump		goto L	next execute the instruction labeled L
Conditional Jump		iffalse x goto L	if x is false, next execute the instruction labeled L
			push param y to the stack

Three Address Code – implementation details

- All three address instructions are represented as a quadruple of
  - result, operator, lhs, and rhs.



- All TACs in table from previous slide can be represented using this form
  - E.g., assignment: x := y op z
  - E.g., array index: x := y[i]
  - E.g., method call: x := call f N
  - …

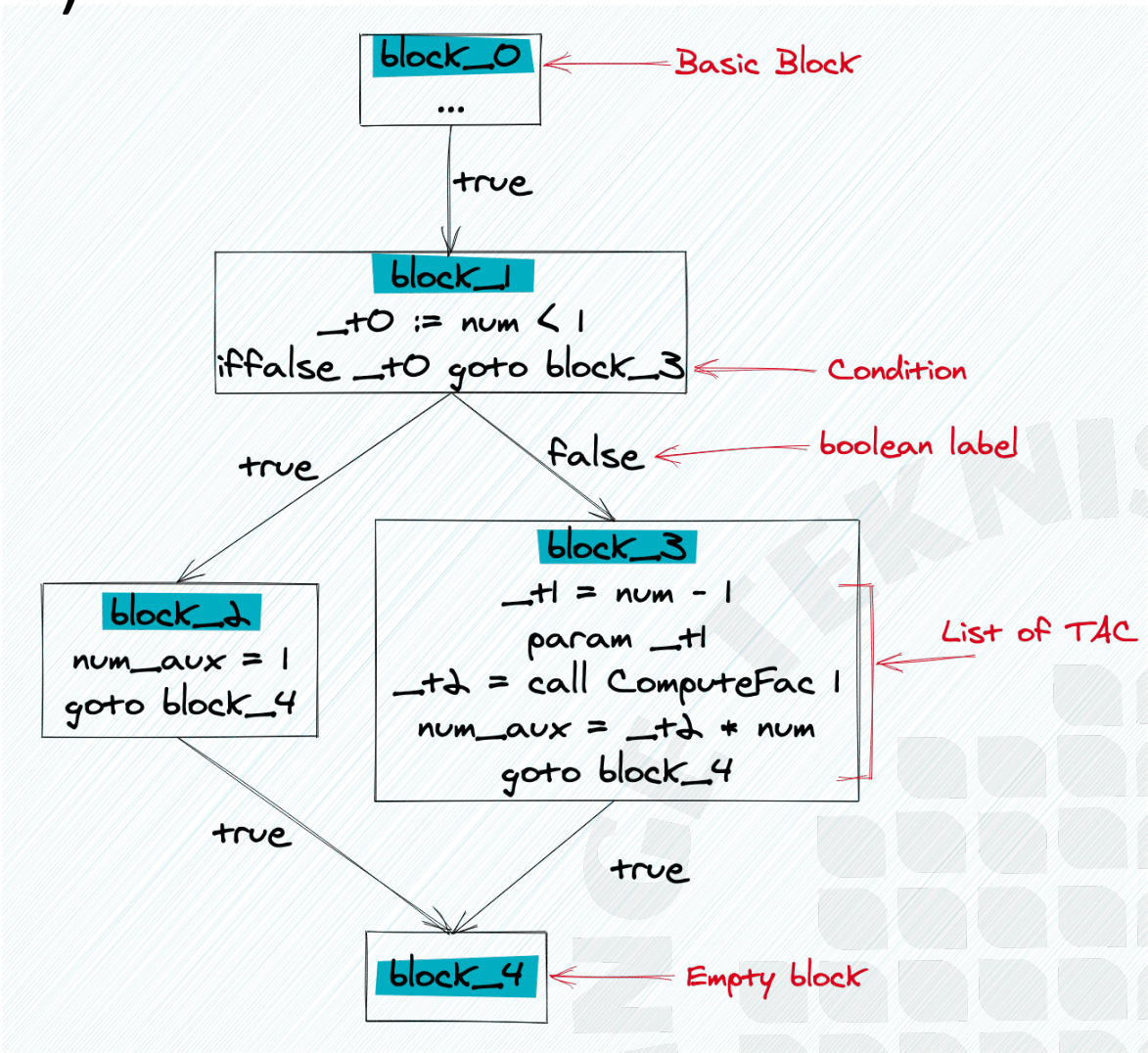
```
class Tac {
  string op, lhs, rhs, result;

  //getters and setters
  dump(){
    printf("%s := %s %s %s", result, lhs, op, rhs);
  }
}

Expression : Tac {
  Expression(string _op, string _y, string _z, string _result)
    : op{_op}, lhs{_y}, rhs{_z}, result{_result} {}
}

MethodCall : Tac {
  MethodCall(string _f, string _N, string result)
    : op{"call"}, lhs{_f}, rhs{_N}, result{_result} {}
}

Jump : Tac {
  Jump(string _label)
    : op{"goto"}, result{_label} {}
}

CondJump : Tac {
  CondJump(string _op, string _x, string _label)
    : op{_op}, lhs{_x}, result{_label} {}
}
...
```

Control Flow Graph (CFG)

- Models the flow of control through program.

- Graph: Basic Block + edges.
  - Edge shows flow of control.
  - Block_0 -> Block_1 means Block_1 may be executed after Block_0.

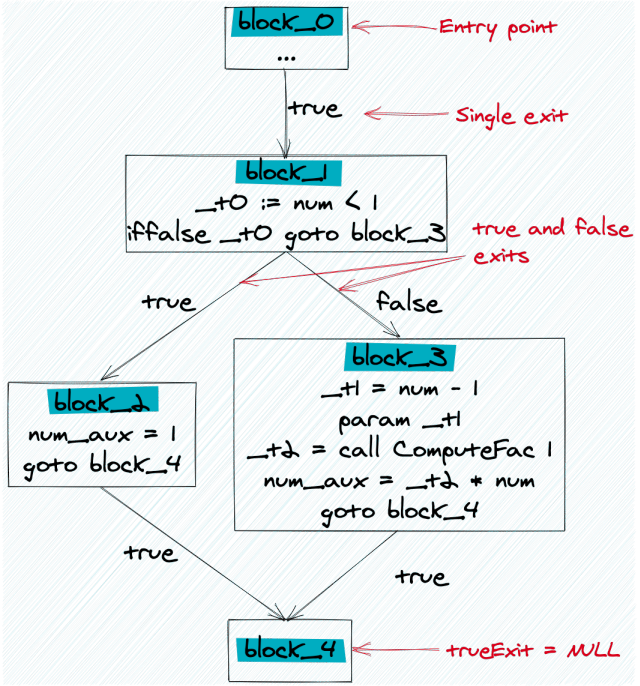
- Basic Block
  - Contains sequence of three address instructions
  - In-edges
    - No in-edge – procedure entry
    - Single-multiple in-edge
  - Out-edges
    - No out-edges - termination
    - Single out-edge – control must flow
    - Two out-edges – control flow by condition



Control Flow Graph (CFG) – implementation details

- Similar to a tree
- Edges in the graph become pointers
  - E.g., trueExit and falseExit
  - trueExit==NULL – no true exit
- Convention
  - Single exit arrow – trueExit
- Entry point – similar to root in the AST

```
class BBlock {
  string name; //unique name
  list<Tac> tacInstructions;
  Tac condition;
  BBlock *trueExit, *falseExit;
  BBlock() : trueExit(NULL), falseExit(NULL) {}
}
BBlock *entry;
```




```
class Tac {
  string op, lhs, rhs, result;

  //getters and setters
  dump(){
    printf("%s := %s %s %s", result, lhs, op, rhs);
  }
}

Expression : Tac {
  Expression(string _op, string _y, string _z, string _result)
    : op{_op}, lhs{_y}, rhs{_z}, result{_result} {}
}

MethodCall : Tac {
  MethodCall(string _f, string _N, string result)
    : op{"call"}, lhs{_f}, rhs{_N}, result{_result} {}
}

Jump : Tac {
  Jump(string _label)
    : op{"goto"}, result{_label} {}
}

CondJump : Tac {
  CondJump(string _op, string _x, string _label)
    : op{_op}, lhs{_x}, result{_label} {}
}
...
```

Translation of AST into IR

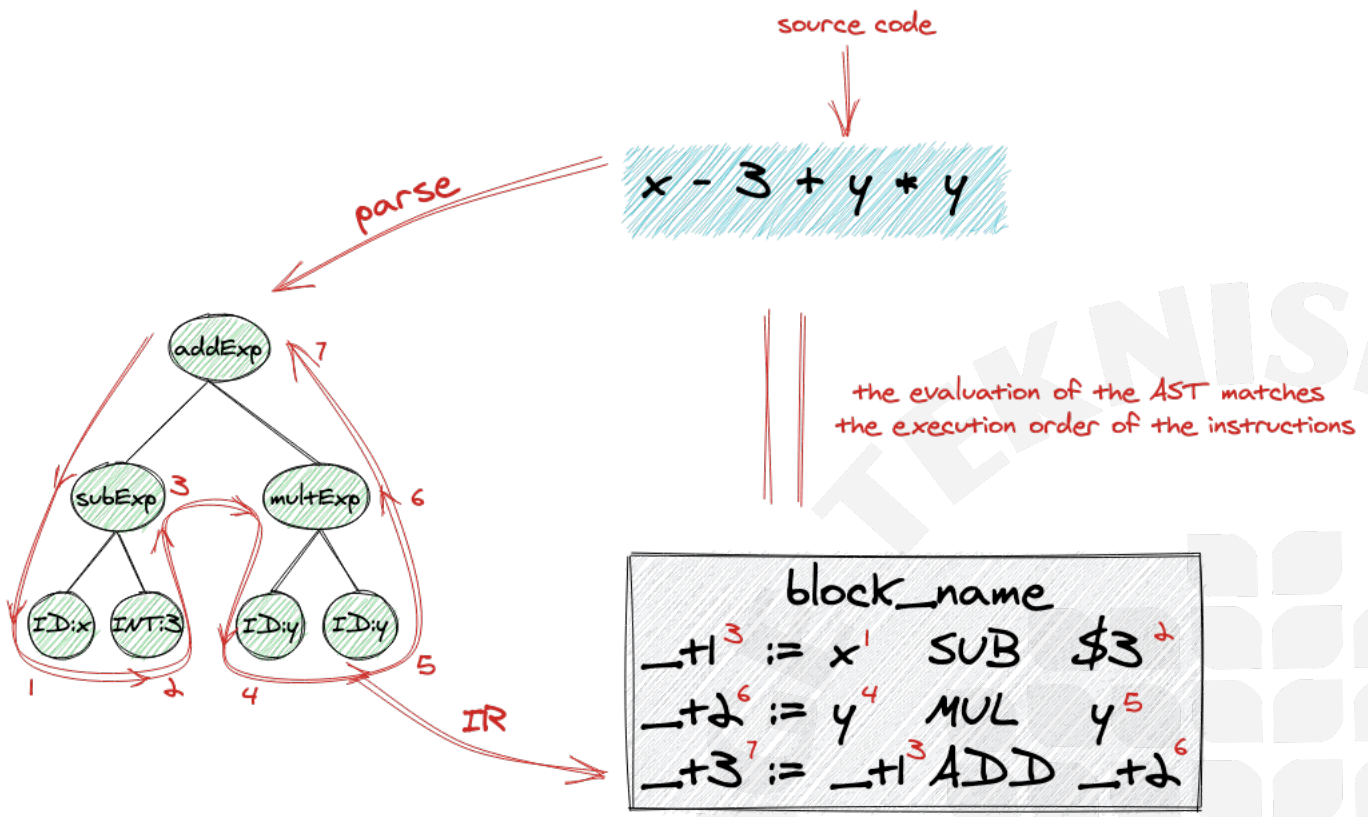
- Each CFG block contains a sequence of three address instructions

- Translating expressions
  - May generate multiple three address instructions
  - May require temporary variables
  - Do not generate new blocks in the CFG

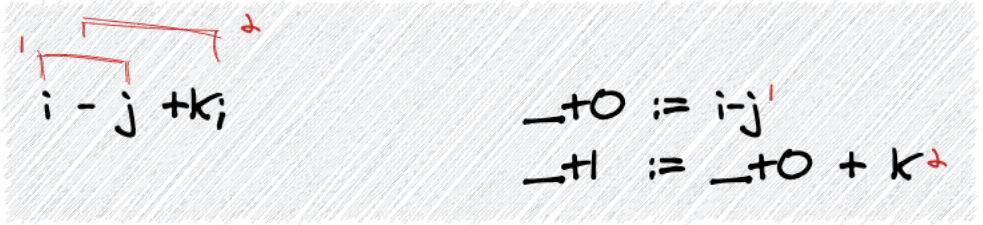
- Translating statements
  - May generate new blocks in the CFG

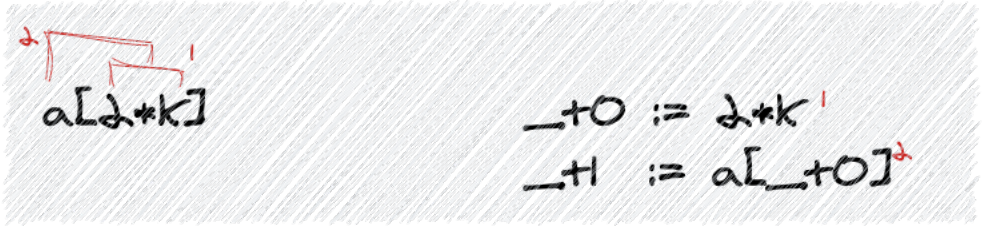
Translating expressions

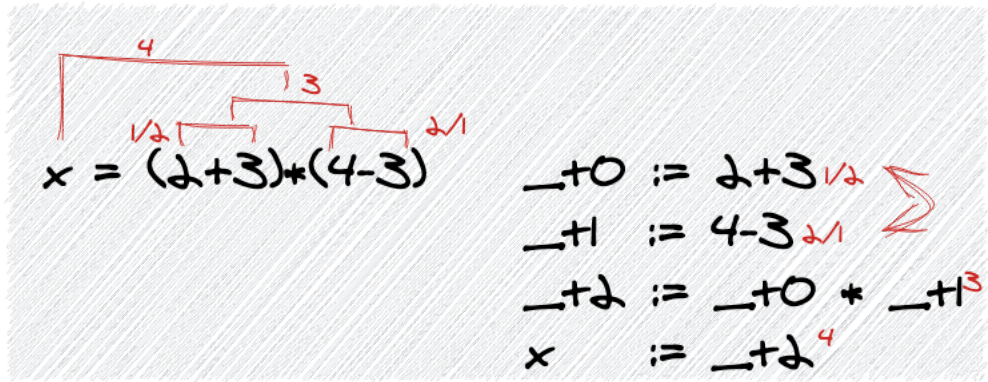
- Post-order traversal of the AST
- May generate multiple three address instructions
- May require temporary variables
- Do not generate new blocks in the CFG
- The evaluation order of the AST matches the execution order of instructions
  - Instructions are executed one after the other

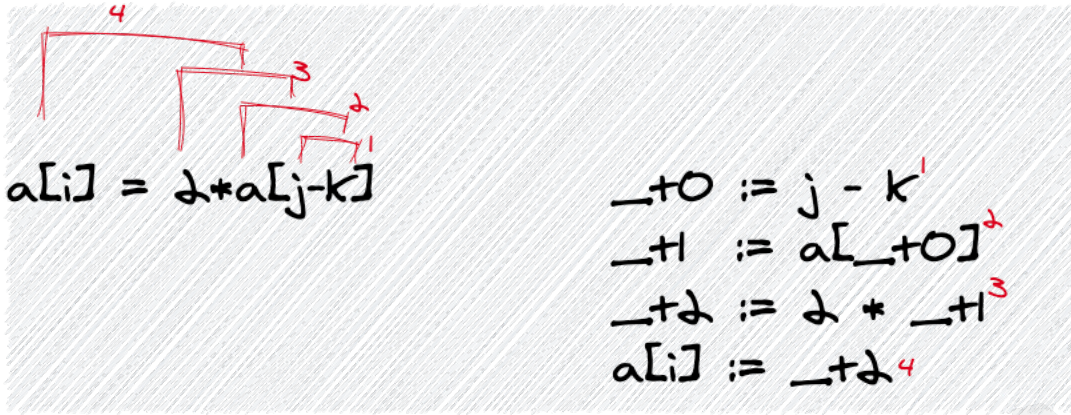


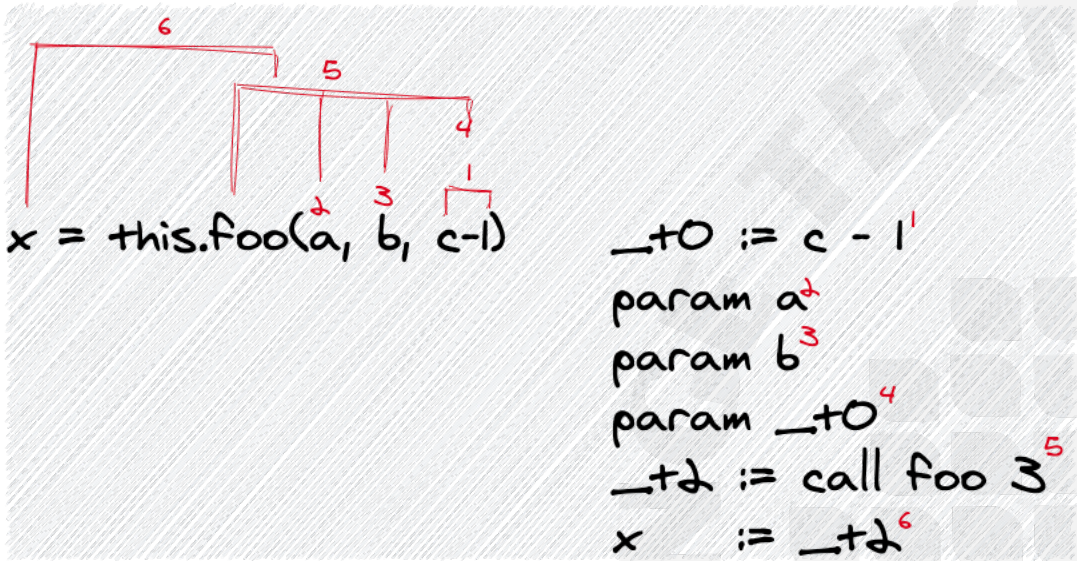
Translating expressions - examples











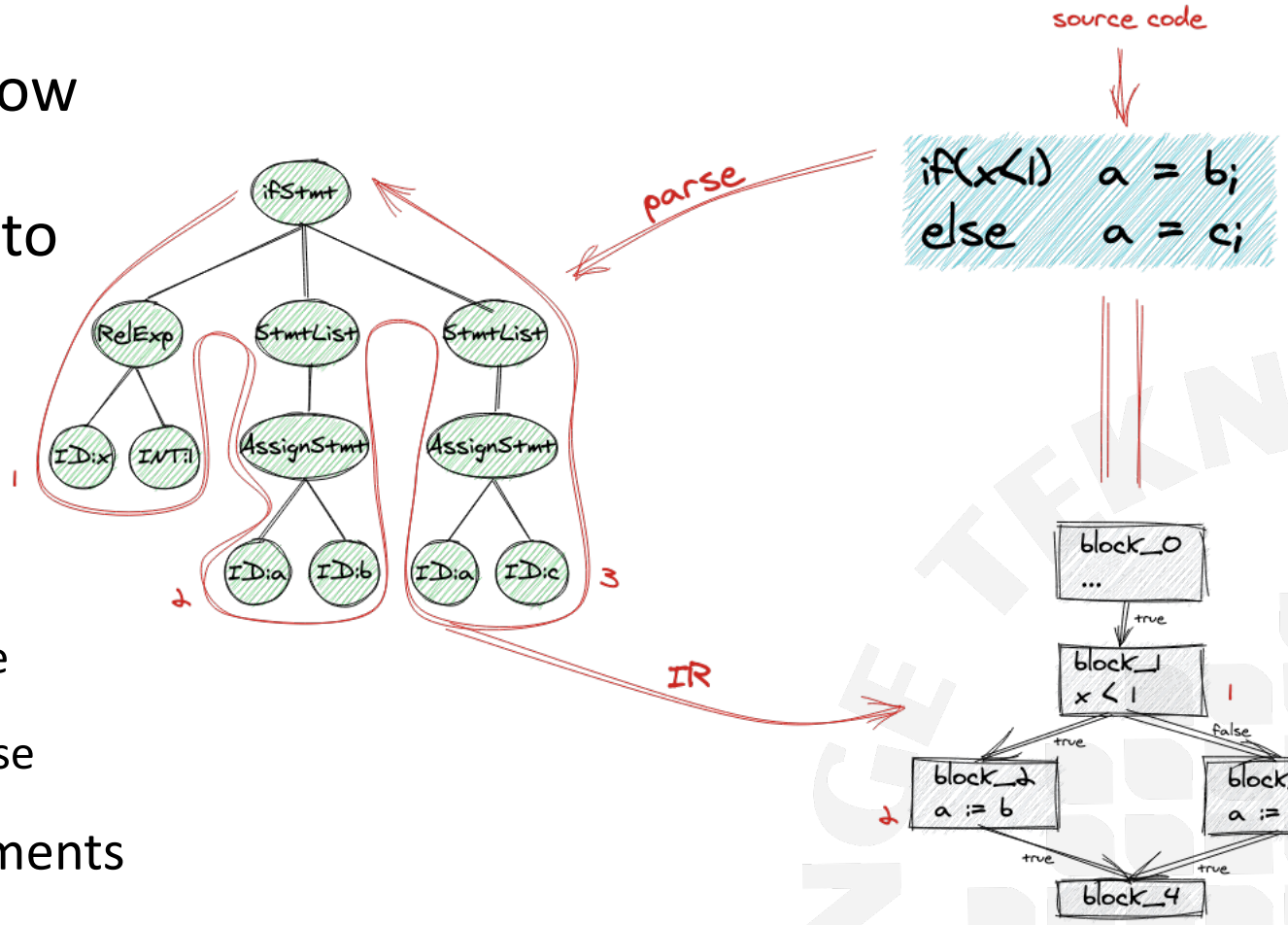
Translating Expressions – code examples

- Arithmetic Expressions and Logic expressions are very similar
  - The only difference is the operator
- Unary operators do not need a rhs
  - E.g., x = !y; -> x := op y
- Identifiers, integers, Booleans, … (leaf nodes)
  - Simply return their values
- Non-leaf nodes need to call the
  - Call genIR recursively and return a name (string)
- For temporary variables, you need to generate unique names
  - E.g., _t0, _t1, …

```
Node() {
  ...
  genIR(...) {...}
  ...
}
SubExpression :: Node() {
  ...
  genIR(BBlock *currentBlock) {
    name = genName(); //generate a unique name
    lhs_name = child[0].genIR(currentBlock);
    rhs_name = child[1].genIR(currentBlock);
    Tac in = Expression("-", lhs_name, rhs_name, name);
    currentBlock->instructions.push_back(in);
    return name;
  }
...
AddExpression :: Node() {
  ...
  genIR(BBlock *currentBlock) {
    name = genName(); //generate a unique name
    lhs_name = child[0].genIR(currentBlock);
    rhs_name = child[1].genIR(currentBlock);
    Tac in = Expression("+", lhs_name, rhs_name, name);
    currentBlock->instructions.push_back(in);
    return name;
  }
...
Identifier :: Node() {
  ...
  genIR(BBlock *currentBlock) {
    return value; // return the name of the idenfitier
  }

...
Integer :: Node() {
  ...
  genIR(BBlock *currentBlock) {
    return value; // return the value of the integer
  }

}
```

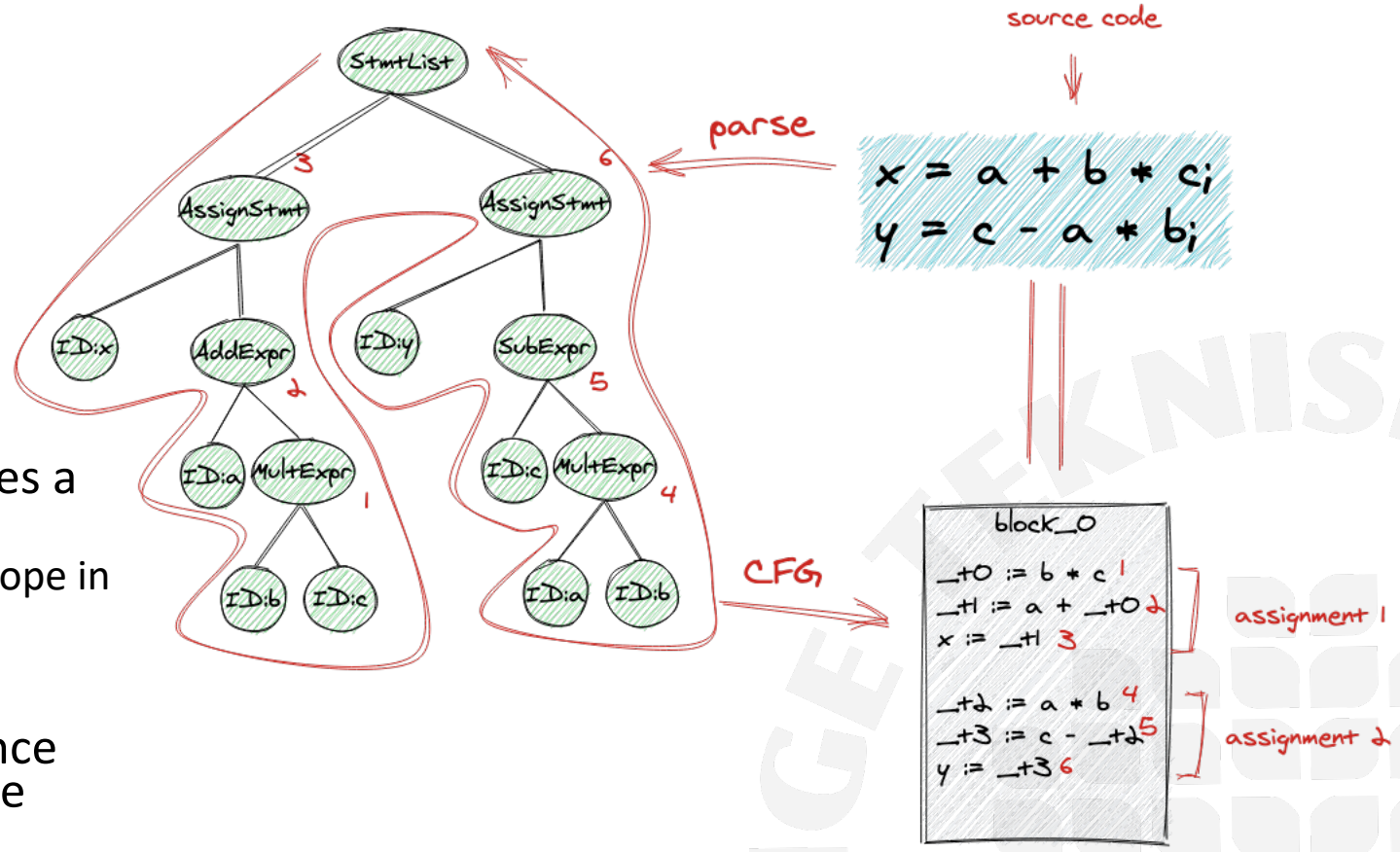
Translating statements

- Statements contain control flow
  - Not all of them
- There exist simple templates to build the control flow graph
  - Assignments
  - Sequence of statements
  - If-else statements
  - While statements
  - Nesting
    - if-else statement inside a while statement
    - while statement inside an if-else statement
  - Method calls and return statements



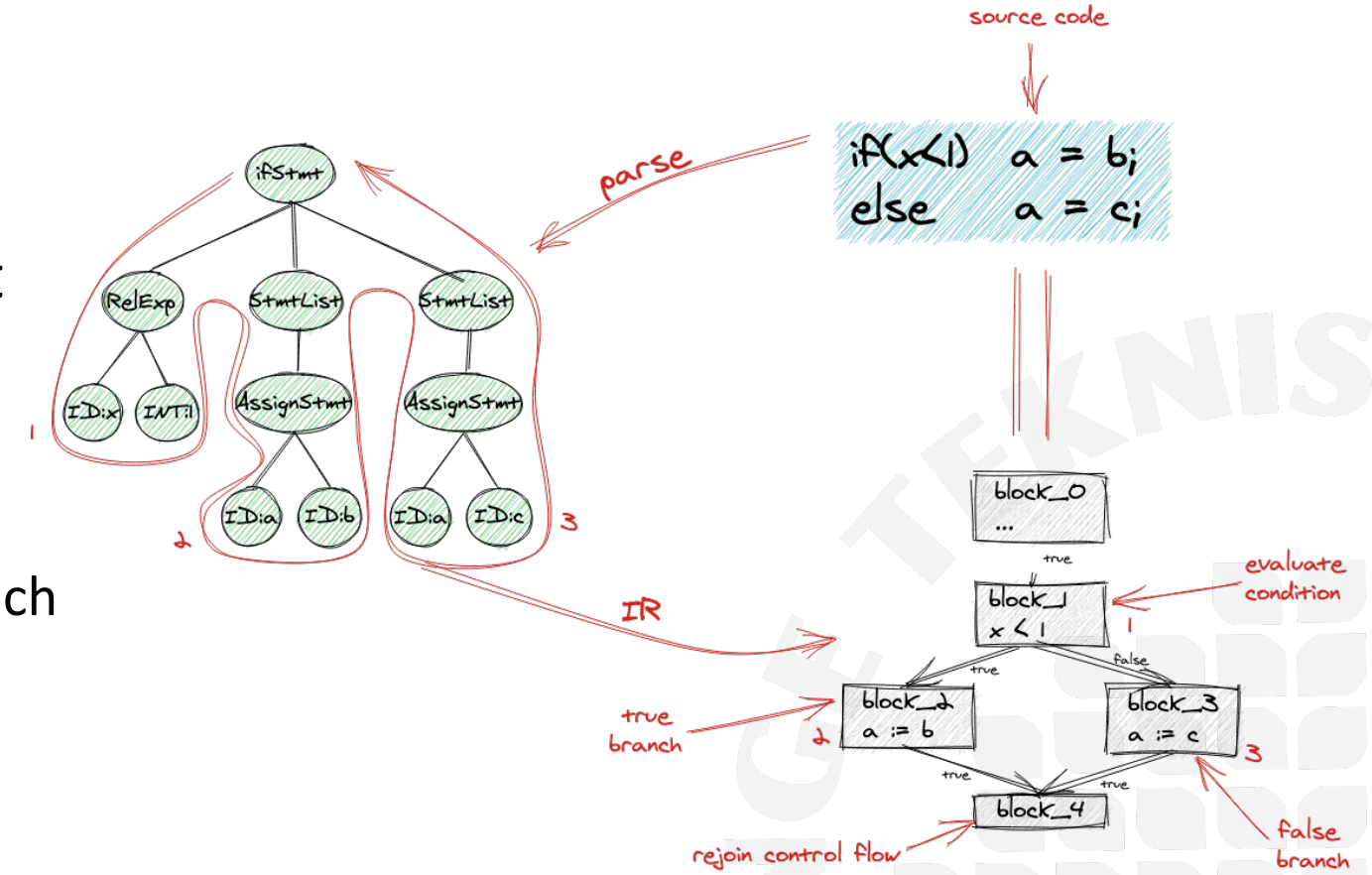
Translating assignment statements

- Simplest form of statements
  - Similar to expressions
- Two steps
  - Evaluate expressions (rhs)
  - Store the result (to the lhs)
- The evaluation of the expression produces a temporary value
  - Add this temporary value to the current scope in the symbol table
- Multiple assignment statements (sequence of assignment) are attached one after the other
  - Do not generate new blocks in the CFG

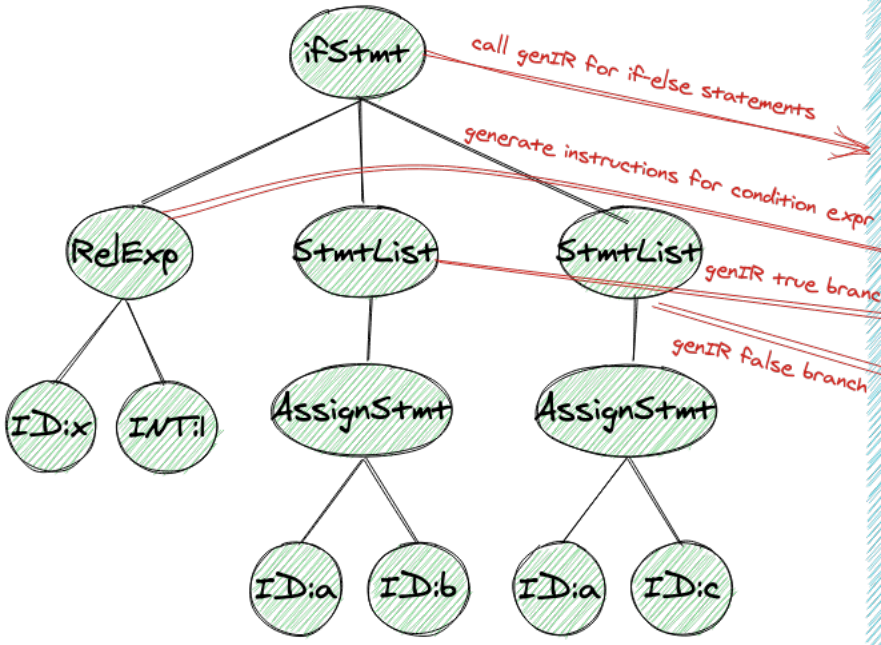


Translating if-else statements

- Encodes run-time decision
- Needs branching
- The evaluation of the boolean expression is done in the current block (block_1)
- Creates two new blocks
  - One for each branch (block_2, block_3)
  - Evaluates the statements inside each block
- Creates another block
  - A rejoinder for both branches (block_4)



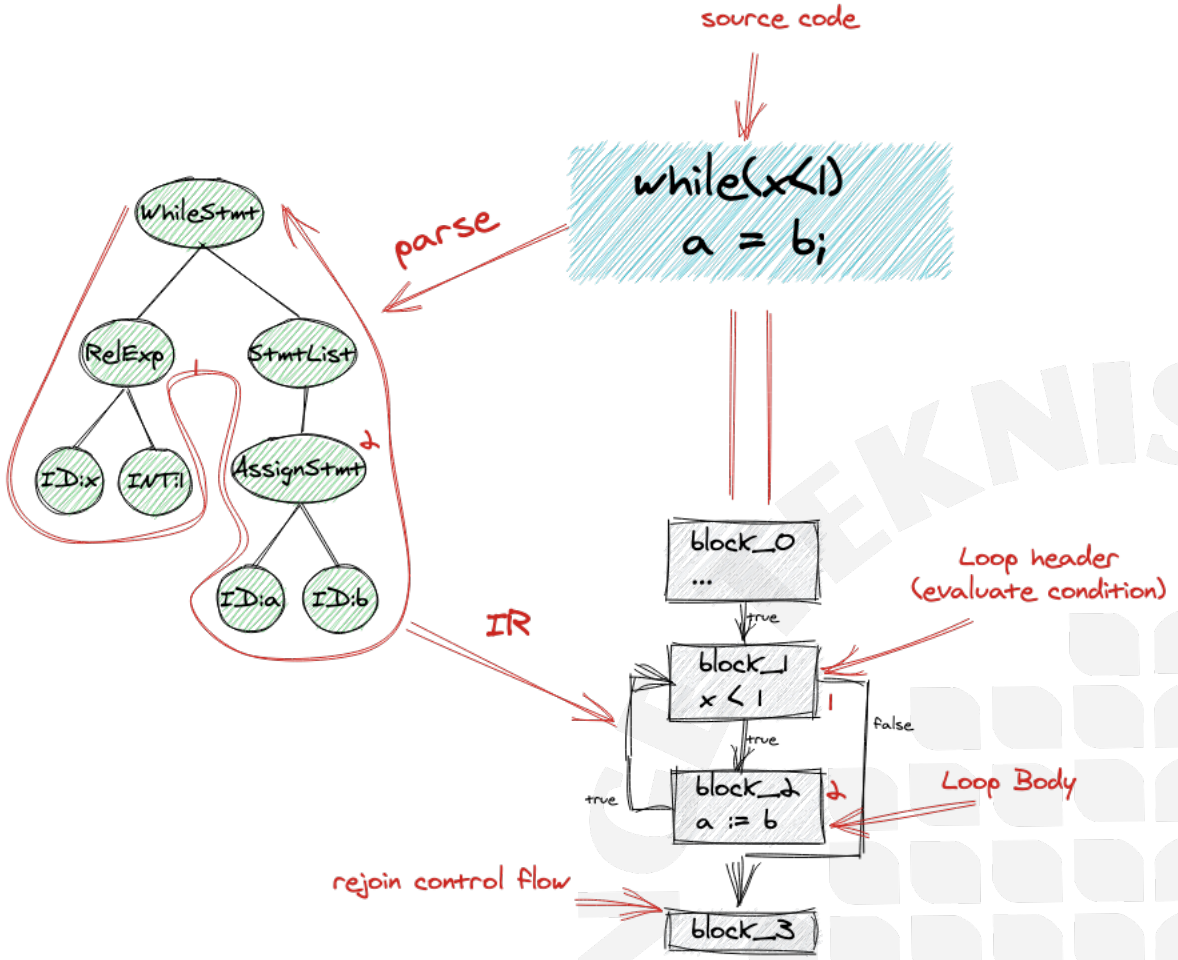
Translating if-else statements - example



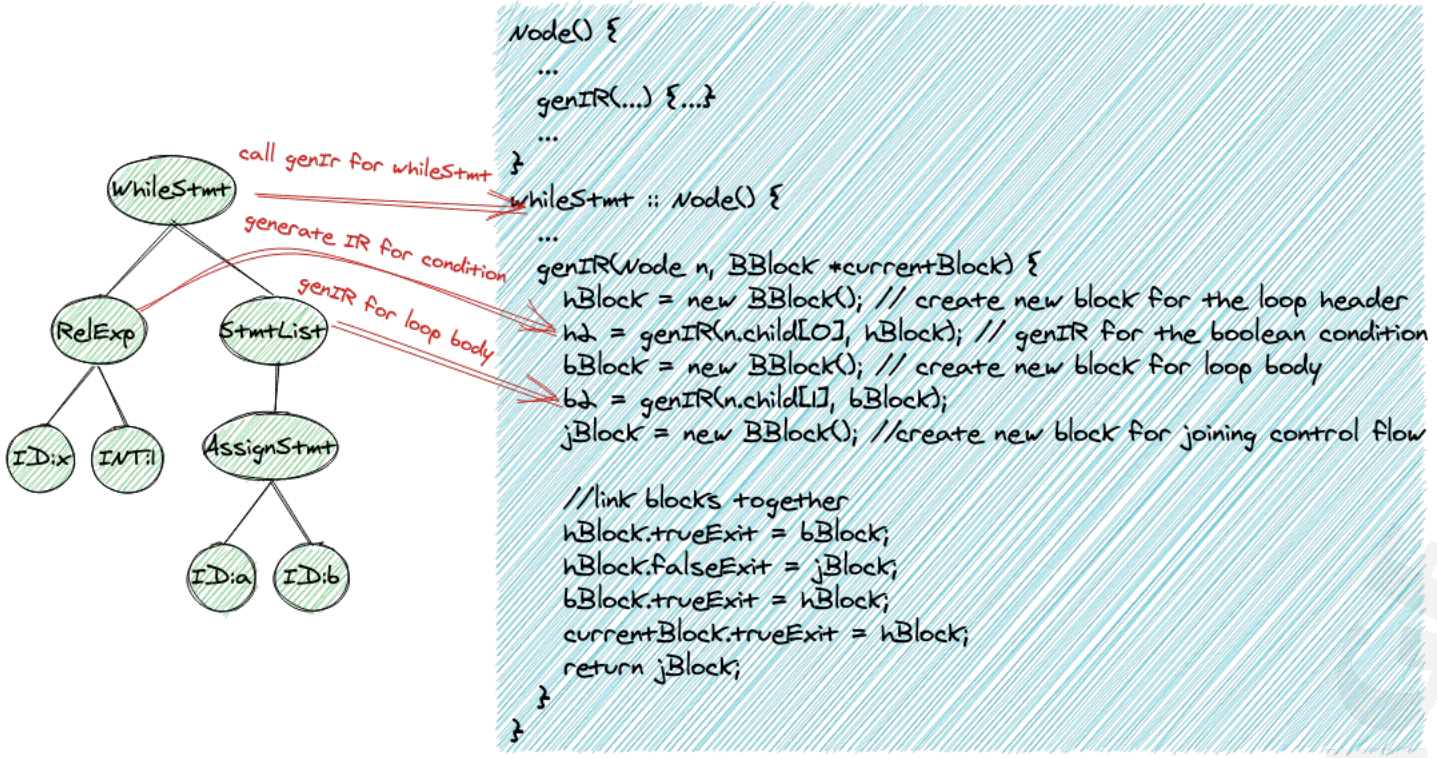

```
Node() {
  ...
  genIR(...) {...}
  ...
}
ifStmt :: Node() {
  ...
  genIR(BBlock *currentBlock) {
    genIR(child[0]); // genIR for the boolean condition
    tBlock = new BBlock(); // create new block for true branch
    t2 = genIR(child[1], tBlock);
    fBlock = new BBlock(); // create new block for false branch
    f2 = genIR(child[2], fBlock);
    jBlock = new BBlock(); //create new block for joining control flow
    //link blocks together
    tBlock.trueExit = jBlock;
    fBlock.trueExit = jBlock;
    currentBlock.trueExit = tBlock;
    currentBlock.falseExit = fBlock;
    return jBlock;
  }
}
```

Translating while statements

- Loop header (block_1)
  - Evaluates the condition
  - Connected to the loop body (block_2) using trueExit arrow,
  - Connected to the exit placeholder (block_3) using falseExit arrow
- Loop body (block_2)
  - List of statements that will be executed while the condition is true
  - Connected to the loop header
- Placeholder (block_3)
  - Rejoins the control flow
  - Filled by following statements

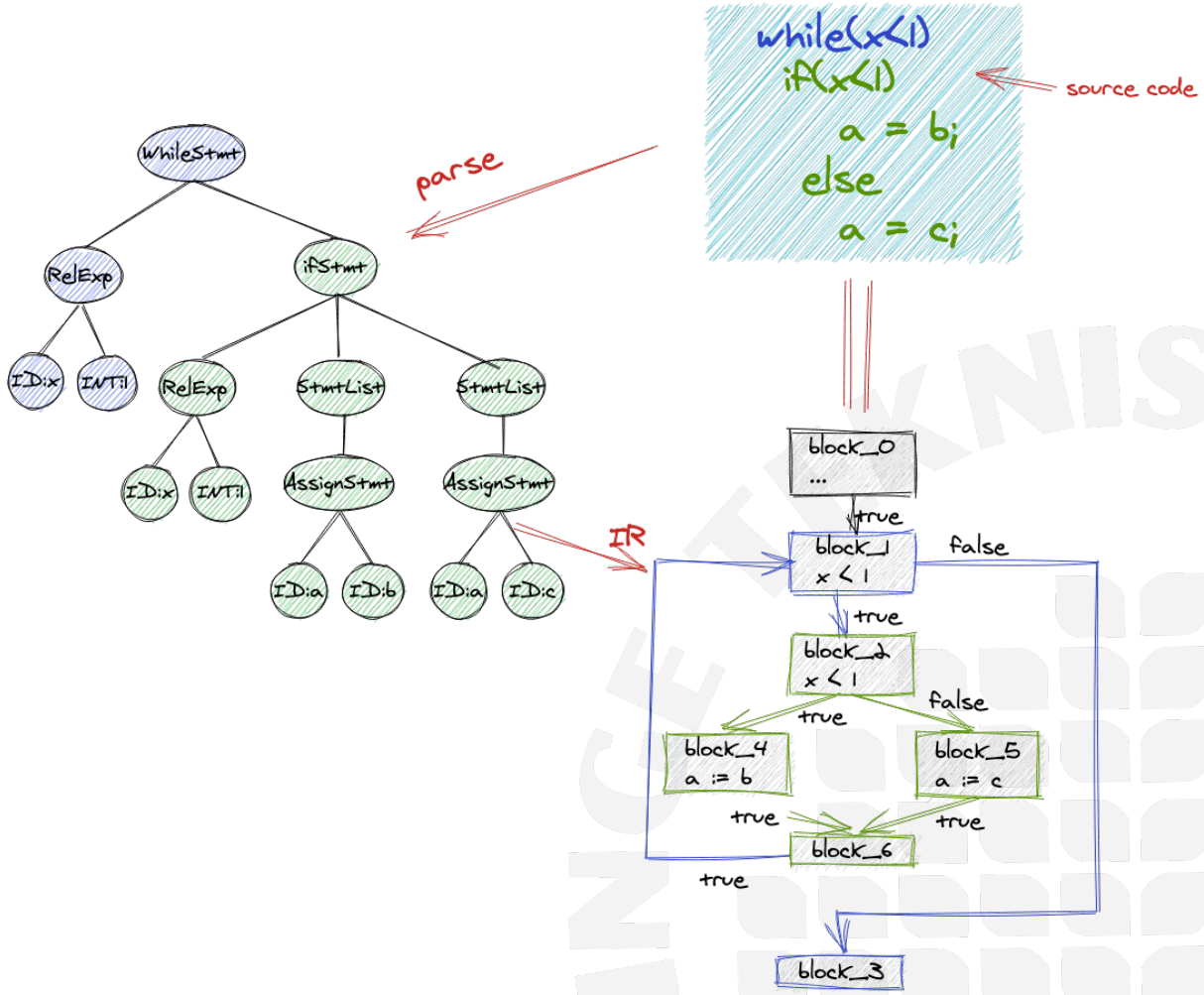


Translating while statements - example



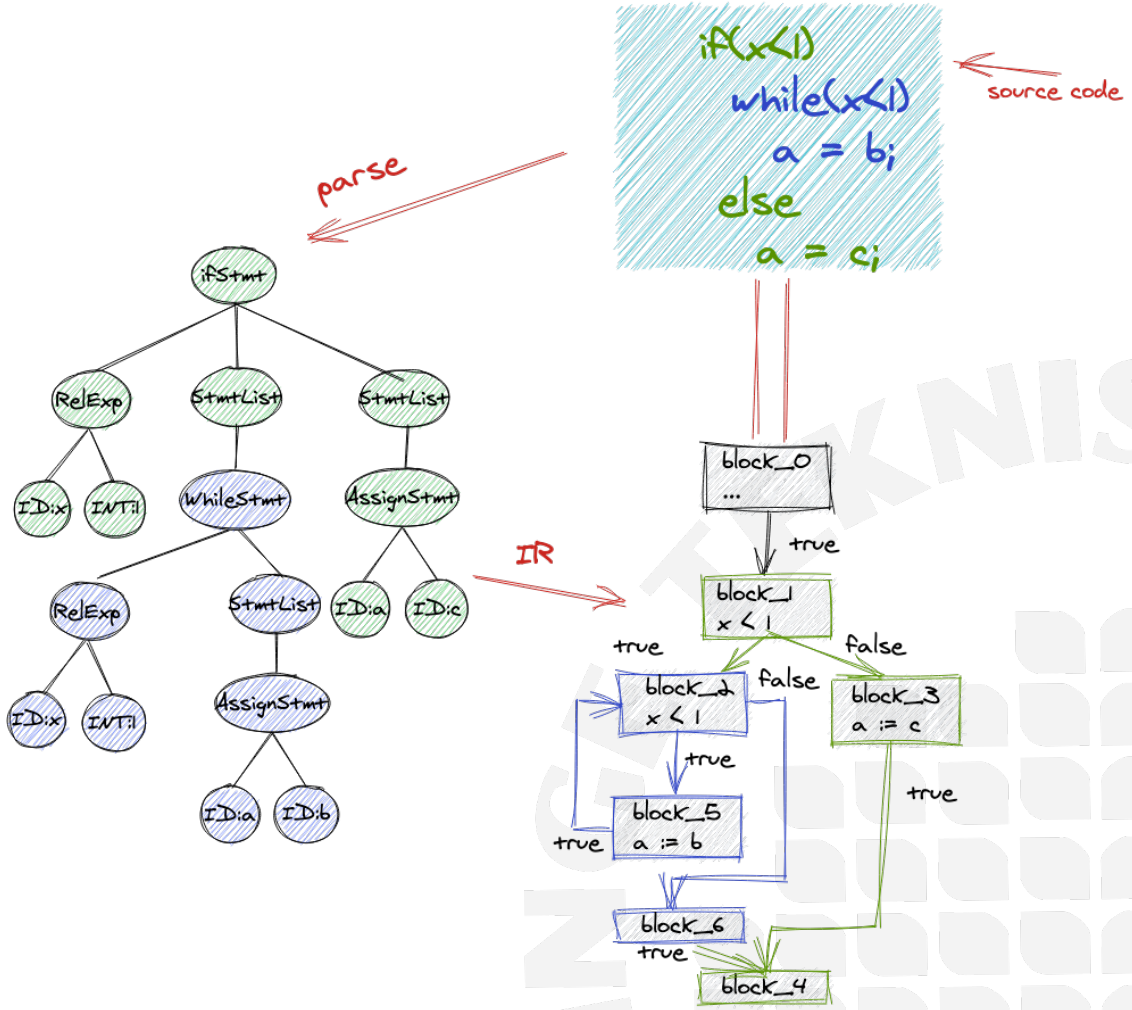
Nesting (if-else inside while statement)

- Combines the templates that we saw in the previous slides
- Two layers
  - Outer part (blue) – the while statement
  - Inner part (green) – the if statement
- Placeholders used for joining the control flow
  - block_2 and block_6



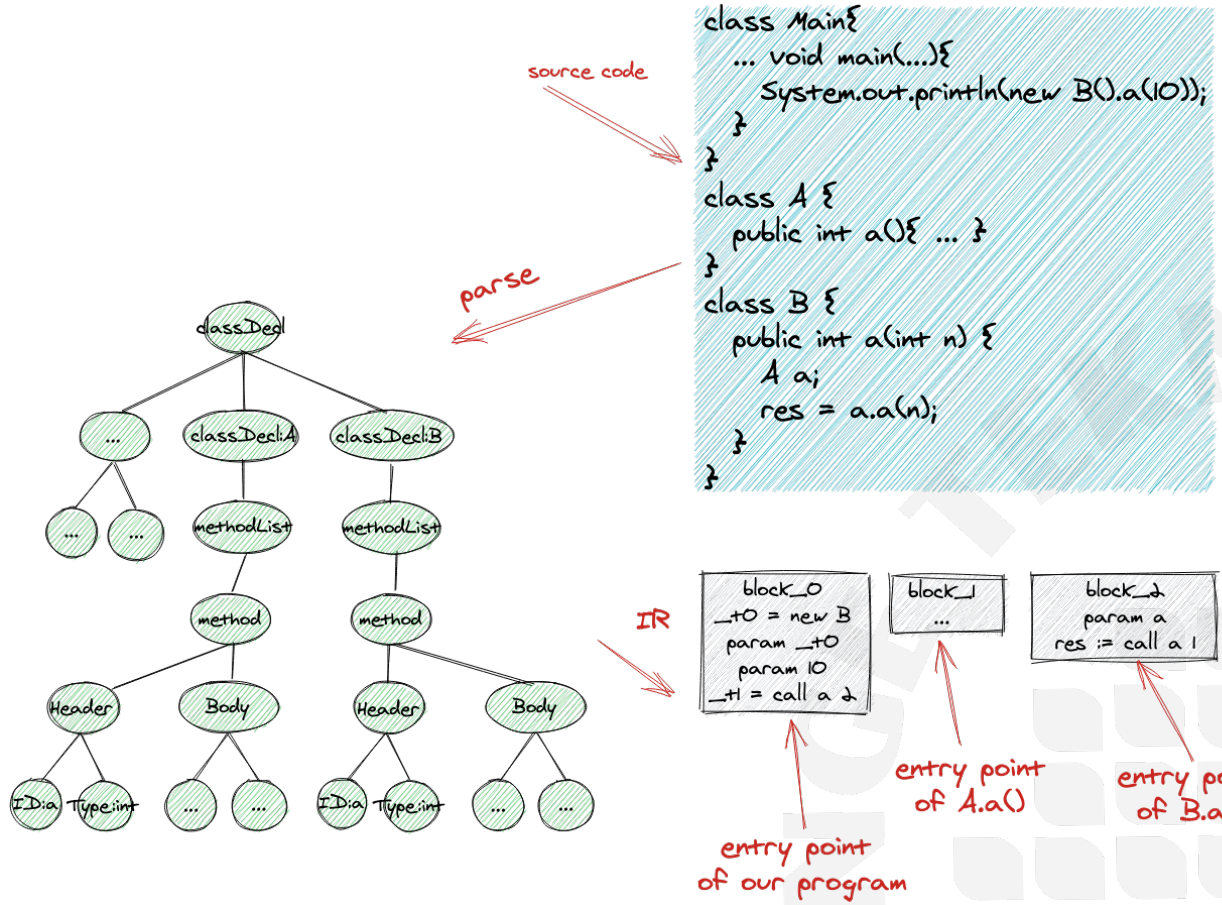
Nesting (while inside if-else statement)

- Like the previous example
  - Combines templates seen before
- Two layers
  - Outer part (green) – The if-else statement
  - Inner part (blue) – the while statement
- The while statement replaces one of the branches (or both) of the if-else statement
- Placeholders used to join the control flow
  - Block_3 and block_6



Methods and method calls

- For each method declaration, a separate CFG is generated
- We need to keep track of which block belongs to which method
- Method calls are simply goto statements
  - “call a 2” in block_0
  - goto block_2



Assignment 3

- Step 1
  - Generate Three Address Code by traversing the AST
- Step 2
  - Generate Java byte-code by traversing the CFG
- Step 3
  - Interpret the Java byte-code