
Norwegian University of
Science and Technology

# Compiler Construction

## Lecture 12: Intermediate representations and three-address code

Michael Engel

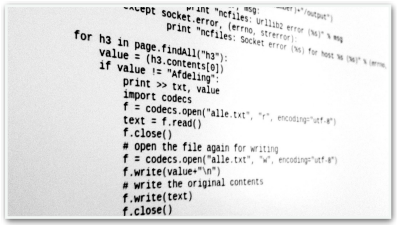
# Overview

- Intro to Intermediate representations
- Classification of IRs
- Graphical IRs: from parse tree to AST
- Linear IRs
  - Example: LLVM IR
- Implementation
  - Three-address code
  - Stack machines
- Hybrid approaches

# What is missing?

Intermediate code

*Source code*




*syntax tree*

*machine-level program*

***Semantic analysis: attributed syntax tree***

- *Name analysis* (check def. & scope of symbols)
- *Type analysis* (check correct type of expressions)
- Creation of *symbol tables* (map identifiers to their types and positions in the source code)

# Code generation

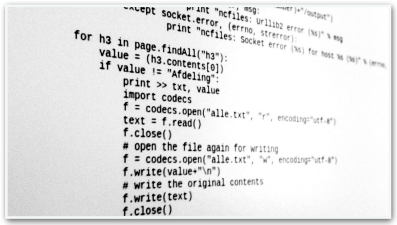
Intermediate code

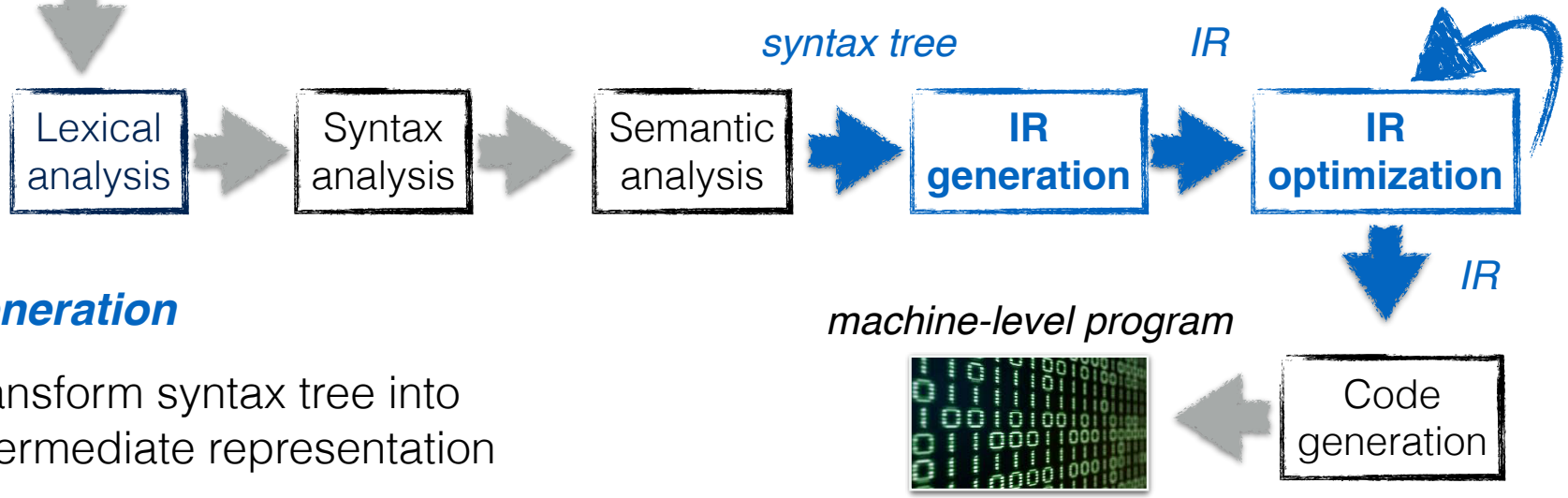
- A syntax tree is a representation of the syntactic structure of a given program
  - we want to ***execute*** the program, i.e. ***control*** and ***data flow***
- Different levels of abstraction required
  - representation for all of the knowledge the compiler derives about the program being compiled
- Most passes in the compiler consume IR
  - the scanner is an exception
- Most passes in the compiler produce IR
  - passes in the code generator can be exceptions
- Many optimizations work for different processors
  - optimizations on IR level can be reused
- IR serves as primary & definitive representation of the code [1]

# A compiler using an IR

Intermediate code

*Source code*




## *IR generation*

- Transform syntax tree into intermediate representation

## *IR optimization*

- Perform generic (non target-specific) optimizations on IR level
- Compilers support many different optimizations, executed in sequence on the IR

# Types of IR

Intermediate code

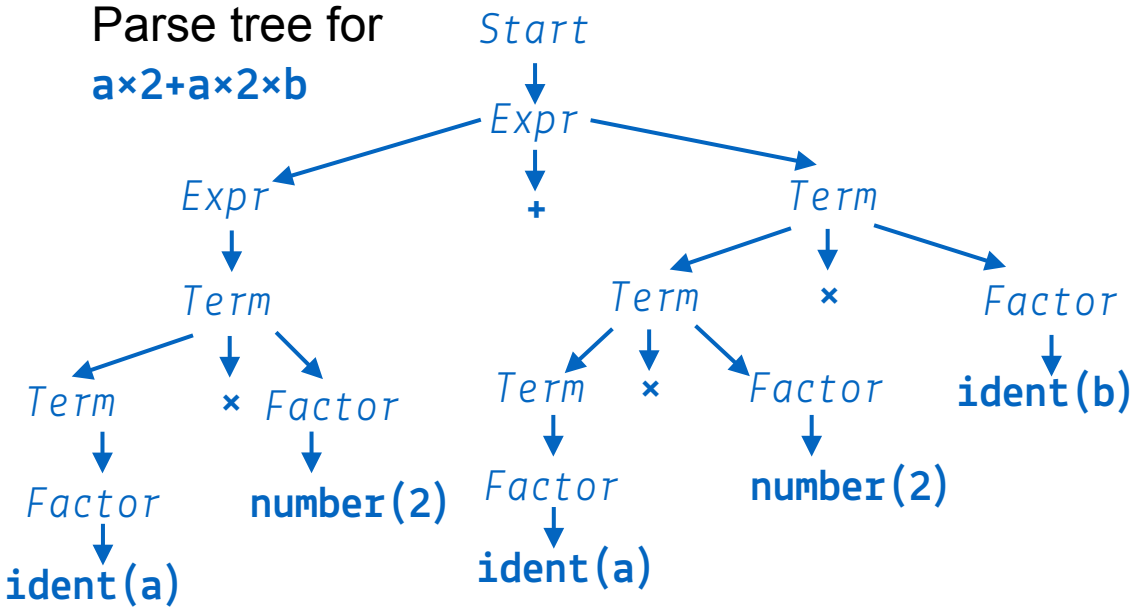
- ***Graphical IRs*** encode the compiler's knowledge in a graph
  - algorithms are expressed in terms of graphical objects: nodes, edges, lists, or trees
  - Our parse trees are a graphical IR
- ***Linear IRs*** resemble pseudo-code for an abstract machine
  - algorithms iterate over simple, linear operation sequences
- ***Hybrid IRs*** combine elements of graphical and linear IRs
  - attempt to capture their strengths and avoid their weaknesses
  - low-level linear IR used to represent blocks of straight-line code and a graph to represent the flow of control

# Graphical IRs: syntax tree → AST

Intermediate code

- So far, we have just talked about syntax trees
  - To be precise, the syntax tree is simply the parse tree generated by the parser
  - The ***abstract*** syntax tree (AST) is an optimized form
    - Uses less memory, faster to process

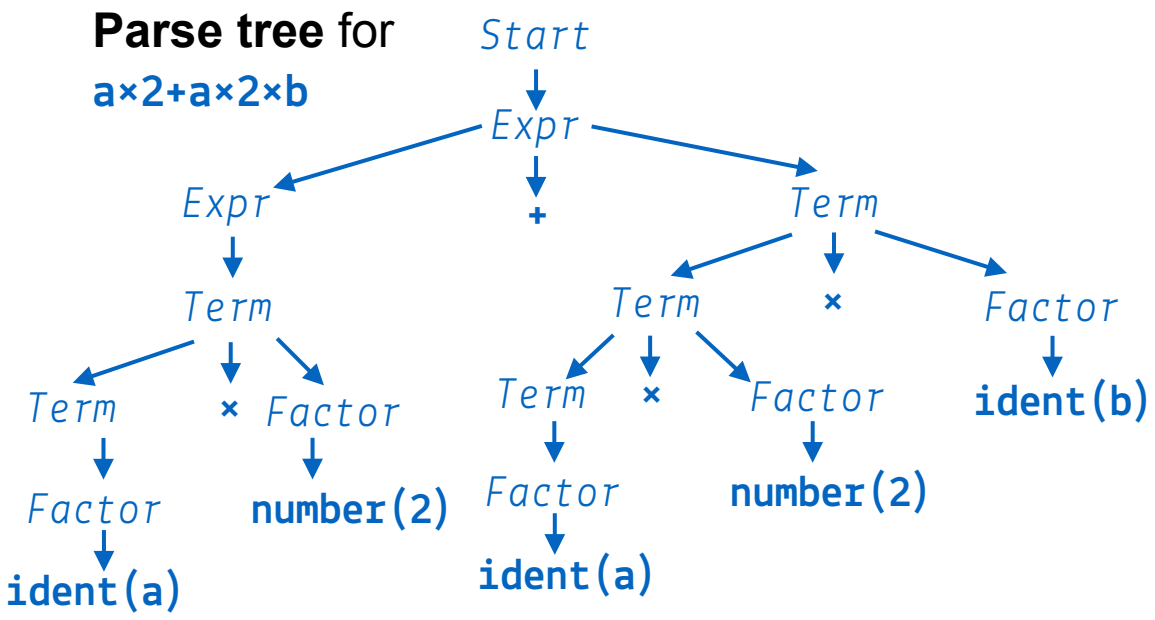
```
1  Start → Expr
2  Expr  → Expr + Term
3        | Expr - Term
4        | Term
5  Term  → Term × Factor
6        | Term ÷ Factor
7        | Factor
8  Factor→ "(" Expr ")"
9        | number
10       | ident
```

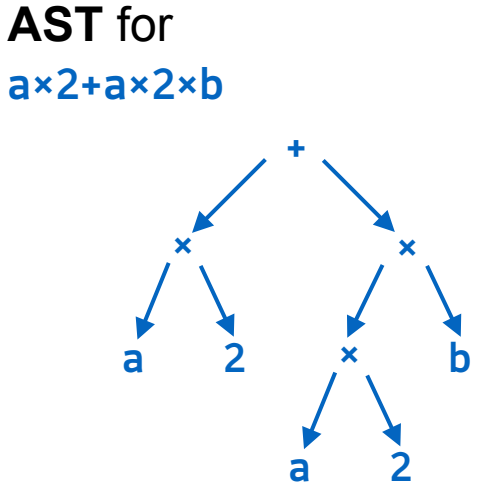


# Graphical IRs: syntax tree → AST

Intermediate code

- The ***abstract*** syntax tree (AST) …
  - retains the essential structure of the parse tree
  - but eliminates the extraneous (nonterminal symbol) nodes
- Precedence and meaning of the expression remain



# From source to machine code level

Intermediate code

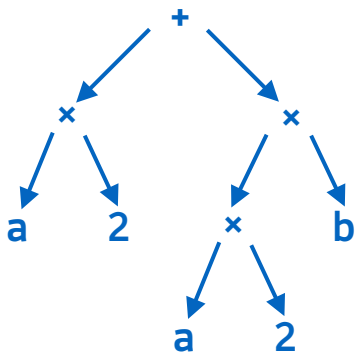
- ASTs are a near-source-level representation
  - Because of its rough correspondence to a parse tree, the parser can built an AST directly
- Trees provide a natural representation for the ***grammatical structure*** of the source code discovered by parsing
  - their rigid structure makes them less useful for representing ***other properties*** of programs
- **Idea:** model these aspects of program behavior differently
  - Different types of IR used in one compiler for different tasks
- Compilers often use more general graphs as IRs
  - Control-flow graphs
  - Dependence graphs

# Directed acyclic graphs (DAGs)

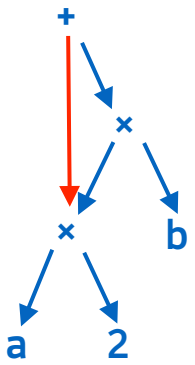
Intermediate code

- DAGs can represent ***code duplications*** in the tree
  - DAG = contraction of the AST that avoids duplications
  - DAG nodes can have multiple parents, identical subtrees are reused
  - sharing makes a DAG more compact than its corresponding AST
- Example: `a×2+a×2×b`
  - Here, the expression "`a×2`" occurs twice
  - DAG can share a single copy of the subtree for this expression
- The DAG ***encodes an explicit hint*** for evaluating the expression:
  - If the value of a cannot change between the two uses of a, then the compiler should generate code to evaluate a×2 once and use the result twice

**AST** for `a×2+a×2×b`




**DAG** for `a×2+a×2×b`

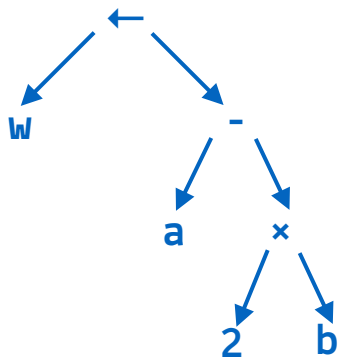


# The level of abstraction

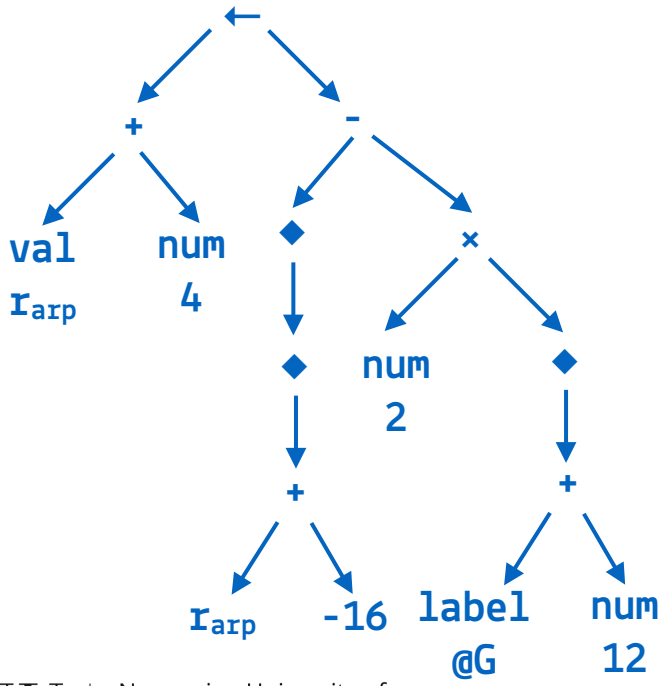
Intermediate code

- Still, the AST here is close to the source code
  - Compilers need additional details, e.g. for tree-based optimization and code generation
  - Source-level tree lacks much of the detail needed to translate statements into assembly code

Source-level **AST** for `w ← a-2×b`




Low-level **AST** for `w ← a-2×b`




Low-level ASTs add this information:

- **`val`** node: value already in a register
- **`num`** node: known constant
- **`lab`** node: assembly-level label
  - typically a relocatable symbol
- ◆: operator that dereferences a value
  - treats value as a memory address and returns the contents of memory at that address (in C: "*" operator)

# Control-flow graphs

Intermediate code

- Simplest unit of control flow in a program is a ***basic block*** (***BB***)
  - maximal length sequence of straightline (branch-free) code
  - sequence of operations that always execute together
    - unless an operation raises an exception
  - control always enters a basic block at its first operation and exits at its last operation
- A ***control-flow graph*** (***CFG***) models the flow of control between the basic blocks in a program
- A CFG is a directed graph, $G = (N, E)$
  - each node $n \in N$ corresponds to a basic block
  - each edge $e = (n_i , n_j) \in E$ corresponds to a possible transfer of control from block $n_i$ to block $n_j$

# CFG example

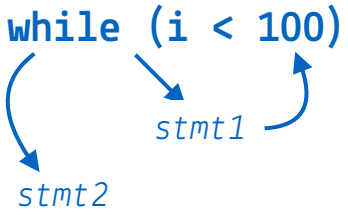
Intermediate code

- ***CFG*** provides a graphical representation of the possible ***runtime*** control-flow paths
  - The CFG differs from ***syntax-oriented IRs***, such as an AST, in which the edges ***show grammatical structure***

The AST for this loop would be acyclic!

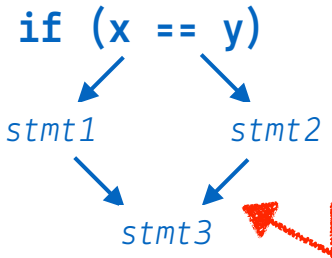
CFG for a while loop:

```
while (i < 100) {
    stmt1;
}
stmt2;
```




CFG for if-then-else:

```
if (x == y) {
    stmt1;
} else {
    stmt2;
}
stmt3;
```




Control always flows from *stmt1* and *stmt2* to *stmt3*

# Use of CFGs

Intermediate code

- Compilers typically use a CFG in conjunction with another IR
  - The CFG represents the relationships among blocks
  - operations inside a block are represented with another IR, such as an expression-level AST, a DAG, or one of the linear IRs.
  - The resulting combination is a hybrid IR
- Many parts of the compiler rely on a CFG, either explicitly or implicitly
  - optimization generally begins with control-flow analysis and CFG construction
  - Instruction scheduling needs a CFG to understand how the scheduled code for individual blocks flows together
  - Global register allocation relies on a CFG to understand how often each operation might execute and where to insert loads and stores for spilled values

# Graphs: dependence graph

Intermediate code

- Compilers also use graphs to encode the flow of values
    - from the point where a value is created, a ***definition*** (***def***)
    - …to any point where it is used, a ***use***
- Data-dependence graph embody this relationship
- ***Nodes*** represent operations
    - Most operations contain both definitions and uses
- ***Edges*** connect two nodes
    - one that defines a value and another that uses it
- Dependence graphs are drawn with edges that run from definition to use

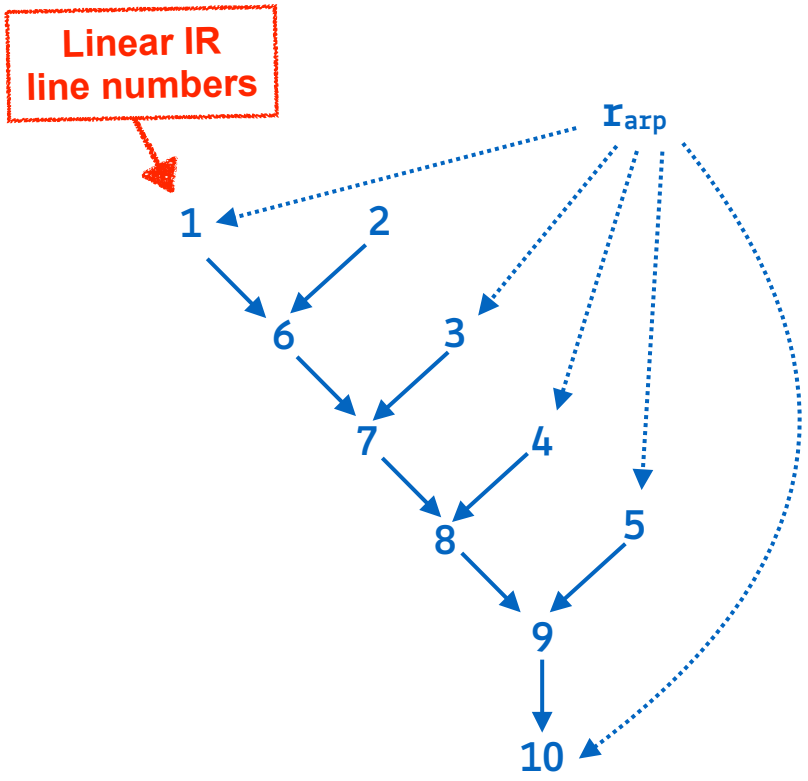
# Dependence graph example

Intermediate code

- To capture the data flow, the dependence graph extracts ***data-flow information*** from an IR representation (here: a linear low-level IR form of a tree)

Linear IR code for `a←a×2×b×c×d`

```
1  load r_arp, @a   => r_a
2  load 2           => r_2
3  load r_arp, @b   => r_b
4  load r_arp, @c   => r_c
5  load r_arp, @d   => r_d
6  mult r_a, r_2    => r_a
7  mult r_a, r_b    => r_a
8  mult r_a, r_c    => r_a
9  mult r_a, r_d    => r_a
10 store r_a        => rarp, @a
```

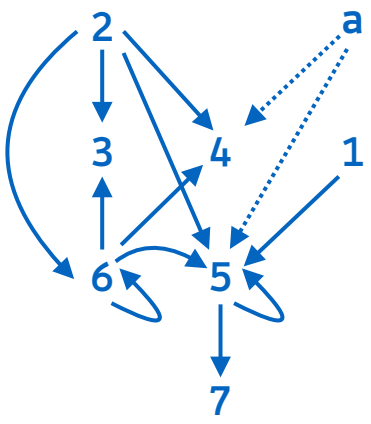


# Interaction: CF and Dependence Graph

Intermediate code

- References to `a[i]` are shown deriving their values from a node representing prior definitions of `a`
- This connects all uses of `a` together through a single node
- Without sophisticated analysis of the subscript expressions, the compiler cannot differentiate between references to individual array elements

```
1 x = 0;
2 i = 1;
3 while (i < 100) {
4   if (a[i] > 0)
5     x = x + a[i];
6   i = i + 1;
  }
7 print(x);
```



# Linear IRs

Intermediate code

**An alternative to graphs**

- A sequence of instructions that execute in their order of appearance
  - linear IRs used in compilers resemble the assembly code for an abstract machine
- Linear IRs must include a mechanism to encode ***transfers of control*** among points in the program
  - control flow in a linear IR usually models the implementation of control flow on the target machine.
  - linear codes usually include conditional branches and jumps
  - control flow demarcates the basic blocks in a linear IR
  - blocks end at branches, at jumps, or just before labelled operations

# Types of linear IRs

Intermediate code

- ***One-address codes*** model the behavior of accumulator machines and stack machines
  - These codes expose the machine's use of implicit names so that the compiler can tailor the code for it
  - The resulting code is quite compact
- ***Two-address codes*** model a machine with destructive operations
  - These codes ***fell into disuse*** as memory constraints became less important; three-address code can model destructive operations explicitly
- ***Three-address codes*** model a machine where most operations take two operands and produce a result
  - The rise of RISC architectures in the 1980s/1990s made these codes popular, since TAC resembles a simple RISC machine

# Linear IRs: stack machines

Intermediate code

- ***Stack-machine code*** offers a compact and storage-efficient representation [3]
  - one form of one-address code
  - assumes the presence of a stack of operands
- Most operations take their operands from the stack and push their results back onto the stack
  - e.g., an integer subtract operation would remove the top two elements from the stack and push their difference onto the stack
- Stack discipline creates a need for some new operations
  - *swap* operation interchanges top two elements of the stack
- Lilith was a stack machine designed at ETHZ for running Modula-2 code [2]

# Example: stack machine code

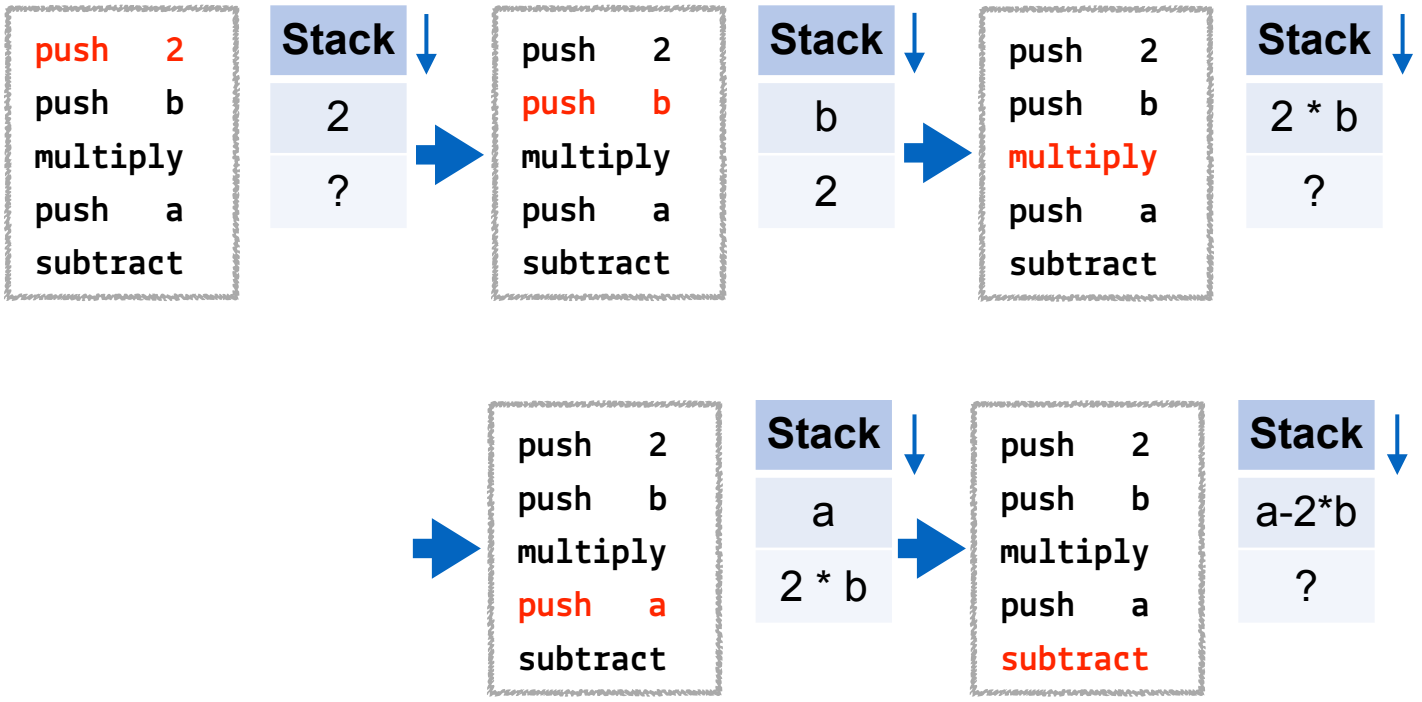
Intermediate code

- Operations remove their operands from stack and push the result
  - Here, the stack grows from the top towards the bottom

Stack machine code for `a - 2 × b`

```
push    2
push    b
multiply
push    a
subtract
```

Execution sequence and related stack state:



# Example: Java Bytecode

Intermediate code

- A compact representation of stack-machine code [3]
  - usually represented in binary form

```
public static void main(String[] args) {
    int a = 1;
    int b = 2;
    int c = a + b;
}
```

```
public static void main(java.lang.String[]);
descriptor: ([Ljava/lang/String;)V
flags: (0x0009) ACC_PUBLIC, ACC_STATIC

Code: stack=2, locals=4, args_size=1

0: iconst_1
1: istore_1
2: iconst_2
3: istore_2
4: iload_1
5: iload_2
6: iadd
7: istore_3
8: return
```

You can disassemble Java bytecode using `javap -v Test.class`

# Three-address code (TAC)

Intermediate code

- Most operations in TAC have the form `i = j op k`
  - one operator (`op`), two operands (`j` and `k`) and one result (`i`)
  - some operators will need fewer arguments
    - e.g. immediate loas and jumps
  - sometimes, an op with more than three addresses is needed
- Three-address code is reasonably compact
  - most ops consist of four items: an operation and three names
  - both the operation and the names are drawn from limited sets
  - operations typically require 1 or 2 bytes
  - names are typically represented by integers or table indices
    - in either case, 4 bytes is usually enough

# TAC example

Intermediate code

- TAC resembles a RISC-like ***register machine***
  - Operands have to be loaded into registers
  - Operations (other than load/store) operate on register values
  - Results are delivered in registers
- Limited constraints for naming/allocating registers compared to real machines

TAC code for `a - 2 × b`

```
t1 ← 2
t2 ← b
t3 ← t1 × t2
t4 ← a
t5 ← t4 - t3
```

ARM assembler code for `a - 2 × b`

```
MOV  R1, #2      // R1=2
LDR  R2, =b
LDR  R2, [R2]    // R2=b
MULU R3, R0, R2  // R3=2*b
LDR  R4, =a
LDR  R4, [R4]    // R4=a
SUB  R5, R4, R3  // R5=R4-R3=a-2*b
```

# Example: LLVM IR

Intermediate code

LLVM IR ("bitcode") is a **typed** TAC [5]

```
define i32 @foo(i32, i32) #0
```

function "foo" gets two int32 params (%0, %1) and returns an int32

```
%3 = alloca i32, align 4
%4 = alloca i32, align 4
```

reserve 2 * 4 bytes memory for temp variables, pointers returned in %3, %4

```
store i32 %0, i32* %3, align 4
store i32 %1, i32* %4, align 4
```

copy %0 → mem @ %3 and %1 → mem @ %4

```
%5 = load i32, i32* %3, align 4
%6 = load i32, i32* %4, align 4
```

mem @ %3 → %5
mem @ %4 → %6

```
%7 = mul nsw i32 2, %6
```

%7 = 2 * %6

```
%8 = sub nsw i32 %5, %7
```

%8 = %5 (=%0) - %7

```
ret i32 %8
```

return %8 to caller

LLVM IR code for

```
int foo(int a, int b) {
  return a - 2 * b;
}
```

```
define i32 @foo(i32, i32) #0 {
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i32 %0, i32* %3, align 4
  store i32 %1, i32* %4, align 4
  %5 = load i32, i32* %3, align 4
  %6 = load i32, i32* %4, align 4
  %7 = mul nsw i32 2, %6
  %8 = sub nsw i32 %5, %7
  ret i32 %8
}
```

Generated with `clang -S -emit-llvm foo.c`

# What's next?

Intermediate code

- More on intermediate representations
  - Efficient implementation
  - Static single assignment (SSA) form

## References


[1] James Stanier and Des Watson. 2013.
Intermediate representations in imperative compilers: A survey.
ACM Comput. Surv. 45, 3, Article 26 (July 2013), https://doi.org/10.1145/2480741.2480743

[2] Richard S. Ohran. 1984.
Lilith: A Workstation Computer for Modula-2. Dissertation ETH No. 7646,
http://www.bitsavers.org/pdf/eth/lilith/ETH7646_Lilith_A_Workstation_Computer_For_Modula-2_1984.pdf

[3] Philip J. Koopman, Jr. 1989.
Stack Computers: the new wave, Ellis Horwood publishers
available at http://users.ece.cmu.edu/~koopman/stack_computers/index.html

[4] Alex Buckley et al. 2014. The Java Virtual Machine Specification, Java SE 8 Edition,
https://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf

[5] LLVM Language Reference Manual. https://llvm.org/docs/LangRef.html