```
 .d8888b.                                  d8b 888                                 .d8888b.   .d8888b.   .d8888b.      d8888
d88P  Y88b                                 Y8P 888                                d88P  Y88b d88P  Y88b d88P  Y88b    d8P888
888    888                                     888                                       888 888    888        888   d8P 888
888         .d88b.  88888b.d88b.  88888b.  888 888  .d88b.  888d888 .d8888b            .d88P 888    888      .d88P  d8P  888
888        d88""88b 888 "888 "88b 888 "88b 888 888 d8P  Y8b 888P"   88K            .od888P"  888    888  .od888P"  d88   888
888    888 888  888 888  888  888 888  888 888 888 88888888 888     "Y8888b.      d88P"      888    888 d88P"      8888888888
Y88b  d88P Y88..88P 888  888  888 888 d88P 888 888 Y8b.     888          X88      888"       Y88b  d88P 888"             888
 "Y8888P"   "Y88P"  888  888  888 88888P"  888 888  "Y8888  888      88888P'      888888888   "Y8888P"  888888888        888
                                  888
                                  888
                                  888
```


Aslan Askarov

aslan@cs.au.dk

2024-08-26

Compilation 2024/Welcome

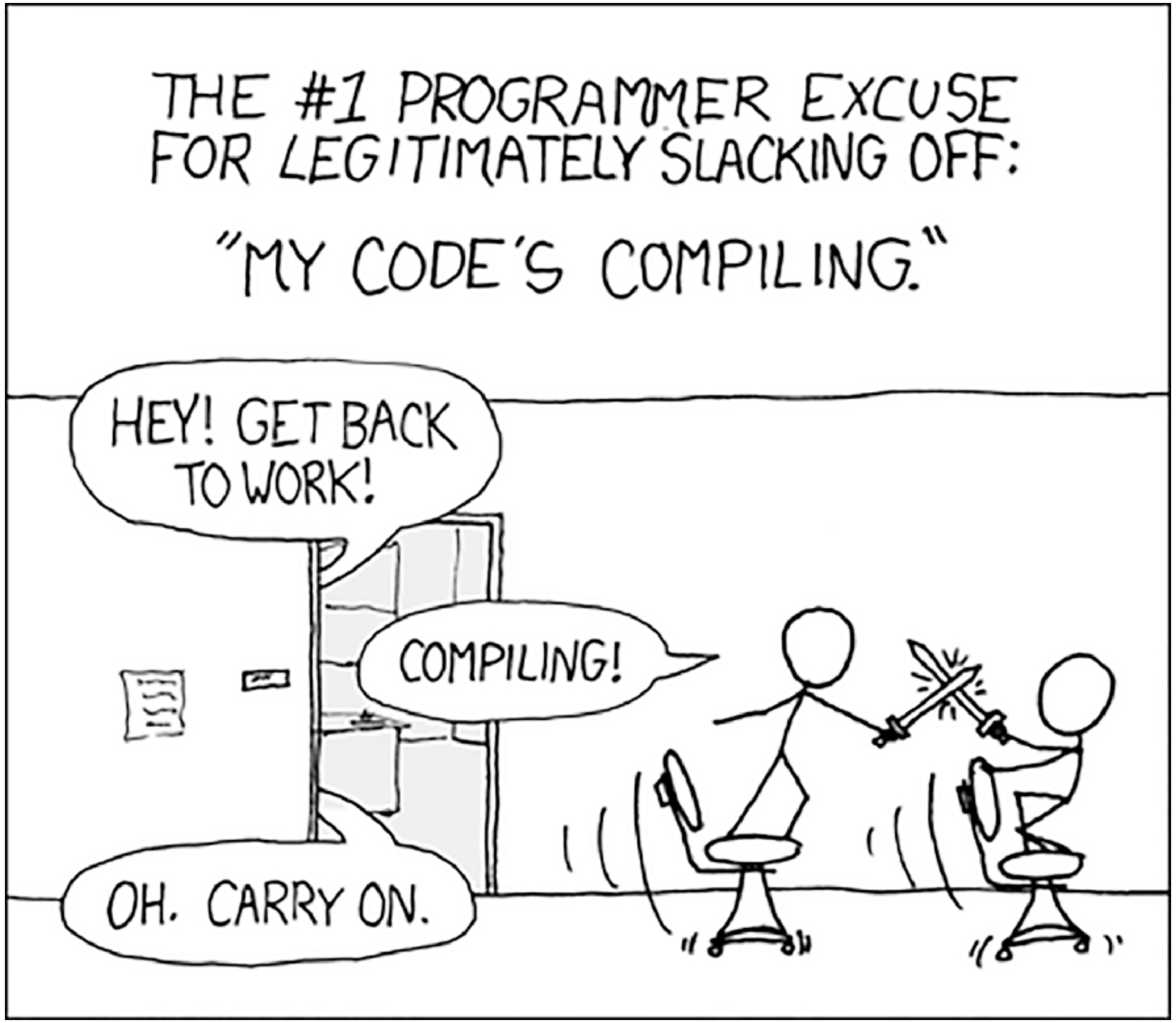
Instructors

- Aslan Askarov
- Amin Timany

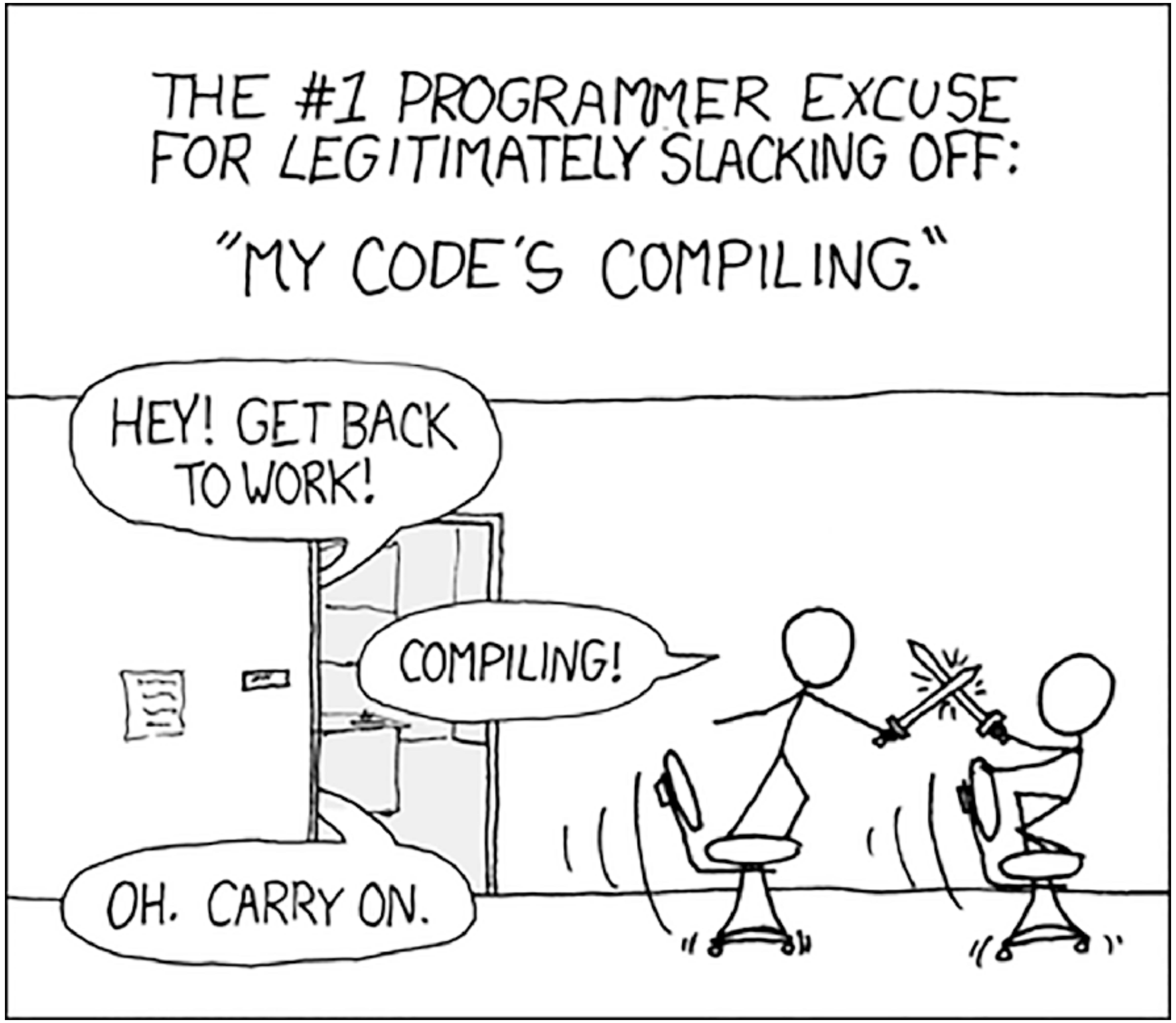
TAs

- Andreas Stenbæk Larsen
- Egor Namakonov
- Eske Hoy Nielsen
- Matthew Christian Demuth Lutze
- Yifan Dong
- Anders Alnor Mathiasen
- Mikael Bisgaard Dahlsen-Jensen

What is a compiler?



What is a compiler?



Go to menti.com and use the code **55949283**

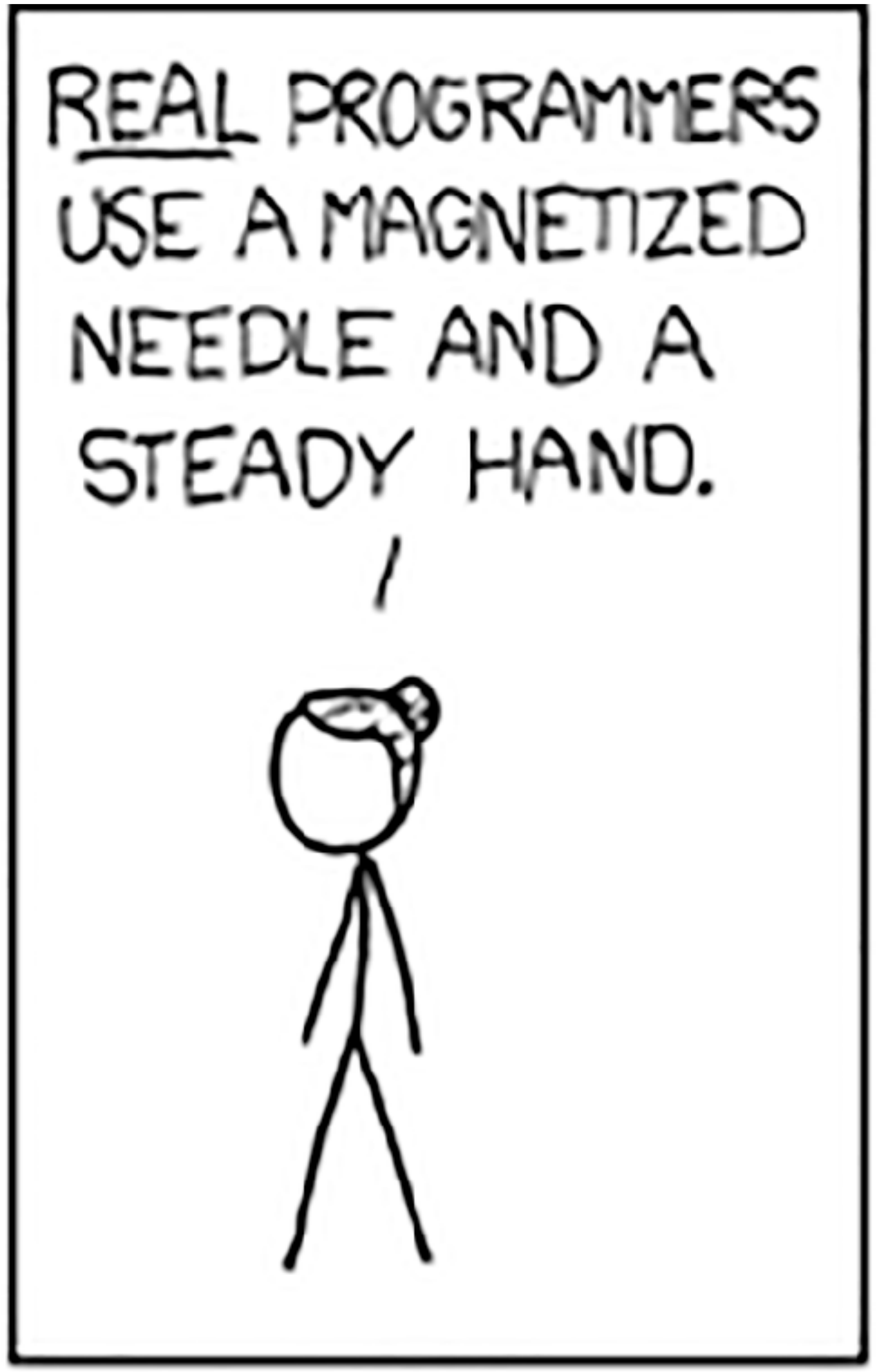
What is a compiler?

- Translator from one programming language (source) into another (target)
  - preserves the semantics
    - the compiler also implicitly defines the semantics, though it's harder to reason about programs with compiler-defined semantics
- Typically:
  - the source language is high-level
  - the target language is low-level
- Not always:
  - Java compiler: Java to interpretable bytecode
  - Java JIT: bytecode to executable

Fundamental properties of a compiler

1. The compiler must preserve the meaning of the program being compiled
2. The compiler must provide some discernible utility

Why use compilers?



Economy

- compilers take care of hundreds of low-level micro decisions that otherwise need to be handled by programmers

Performance

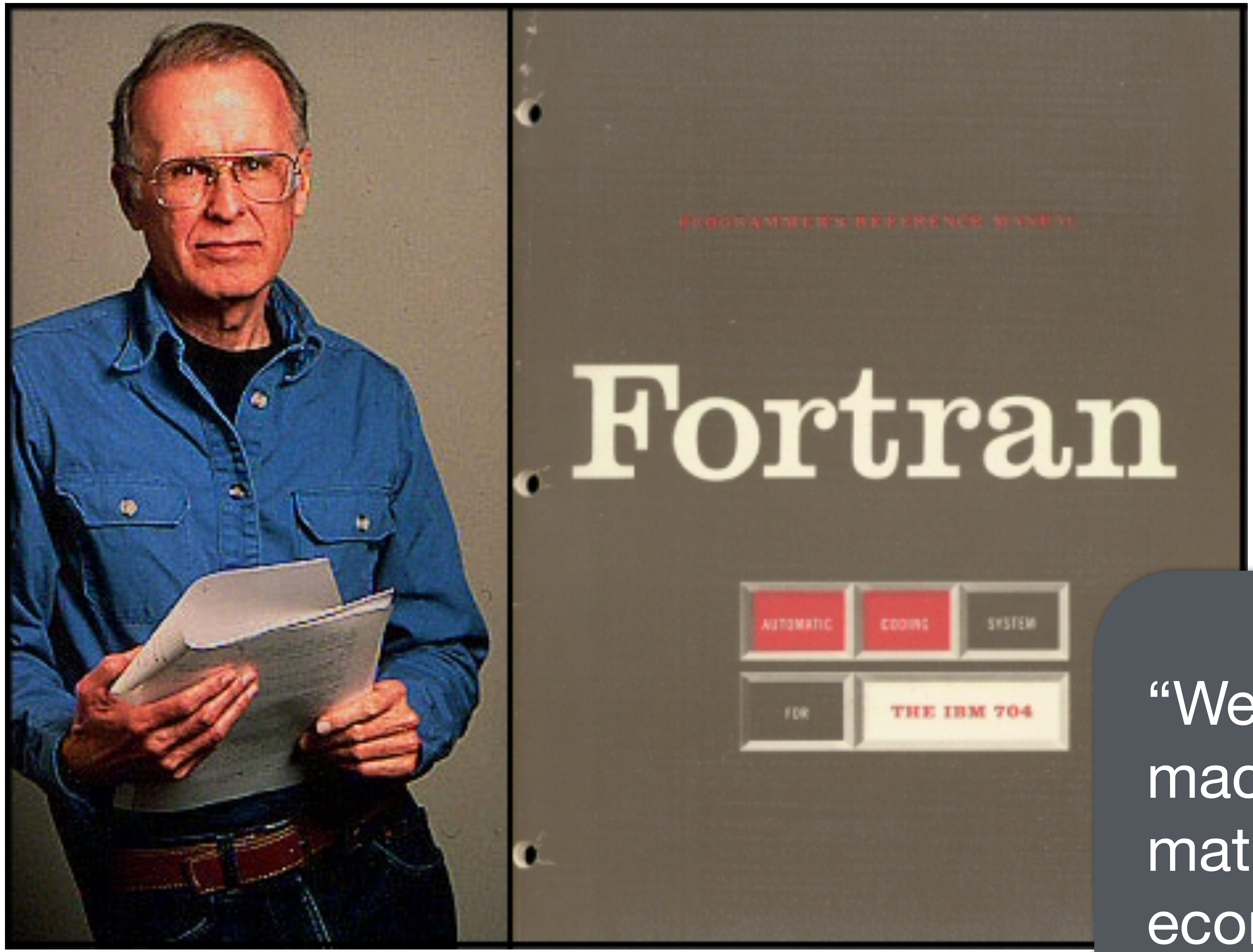
- modern compilers generate better code than most programmers
  - e.g., automatic parallelization on multi-core, loop optimizations

Safety & Security

- compilers implement safety and security checks

Brief history of compilers

1952: Grace Hopper introduces the term “Compiler” for A-0 programming language

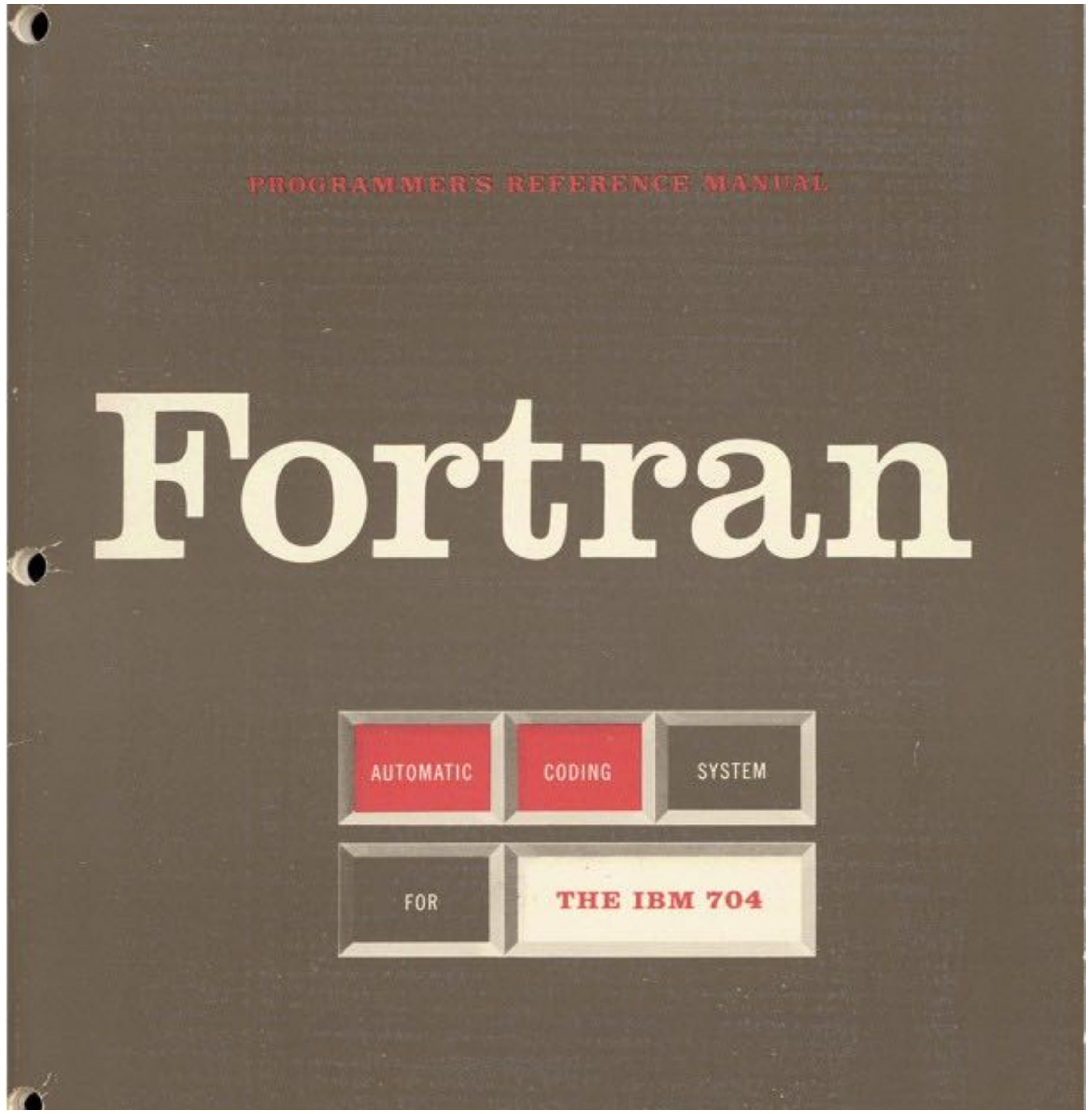


“We went on to raise the question “…can a machine translate a sufficiently rich mathematical language into a sufficiently economical program at a sufficiently low cost to make the whole affair feasible?”

— J. Backus The History of Fortran I, II, and III (1978)

Fortran compiler

- Lead by John Backus at IBM
- Motivated by the economics of programming
- Had to overcome deep skepticism
- Focused on efficiency of the generated code
- Pioneered many concepts and techniques
- Revolutionized computer programming



Brief history (cont'd)

1960 - 1975

- early bootstrapping compilers for LISP
- proliferation of programming languages
- primary focus on just having a programming language rather than efficient code, parsing

1975 - present

- focus on code generation/optimization/paradigms
- the lifetime of generated code increased (as the the number of different machine types decreased)

How good are today’s compilers?

How good are today's compilers?

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

long factorial(long X) {
  if (X == 0) return 1;
  return X*factorial(X-1);
}

int main(int argc, char **argv) {
  printf("%ld\n", factorial(10));
  return 0;
}
```

Source C program

```
…
Ltmp9:
     .cfi_def_cfa_register %rbp
     leaq L_.str(%rip), %rdi
     movl $3628800, %esi          ## imm = 0x375F00
     xorl %eax, %eax
     callq    _printf
     xorl %eax, %eax
     popq %rbp
     ret
     .cfi_endproc

     .section __TEXT,__cstring,cstring_literals
L_.str:                                   ## @.str
     .asciz   "%ld\n"
```

Compiled assembly

$ clang factorial.c -S -O3 -o-

Why study compilers?

"Analysis-synthesis" paradigm

- The concept of analyzing input, constructing semantic representation, and synthesizing output reappears in many problem areas

Judicious use of formalisms

- Front-end: regular expressions and context-free grammars
- Type-checking: constraint solving and inference
- Code generation: pattern matching / dynamic programming

Use of program generating tools

Think like a compiler writer!

Think like a compiler writer?

- Understand implementations of programming language abstractions
  - the cost, the opportunity
- See what/how technical problems can be solved with PL techniques
- Differentiate hype/fluff from novelty

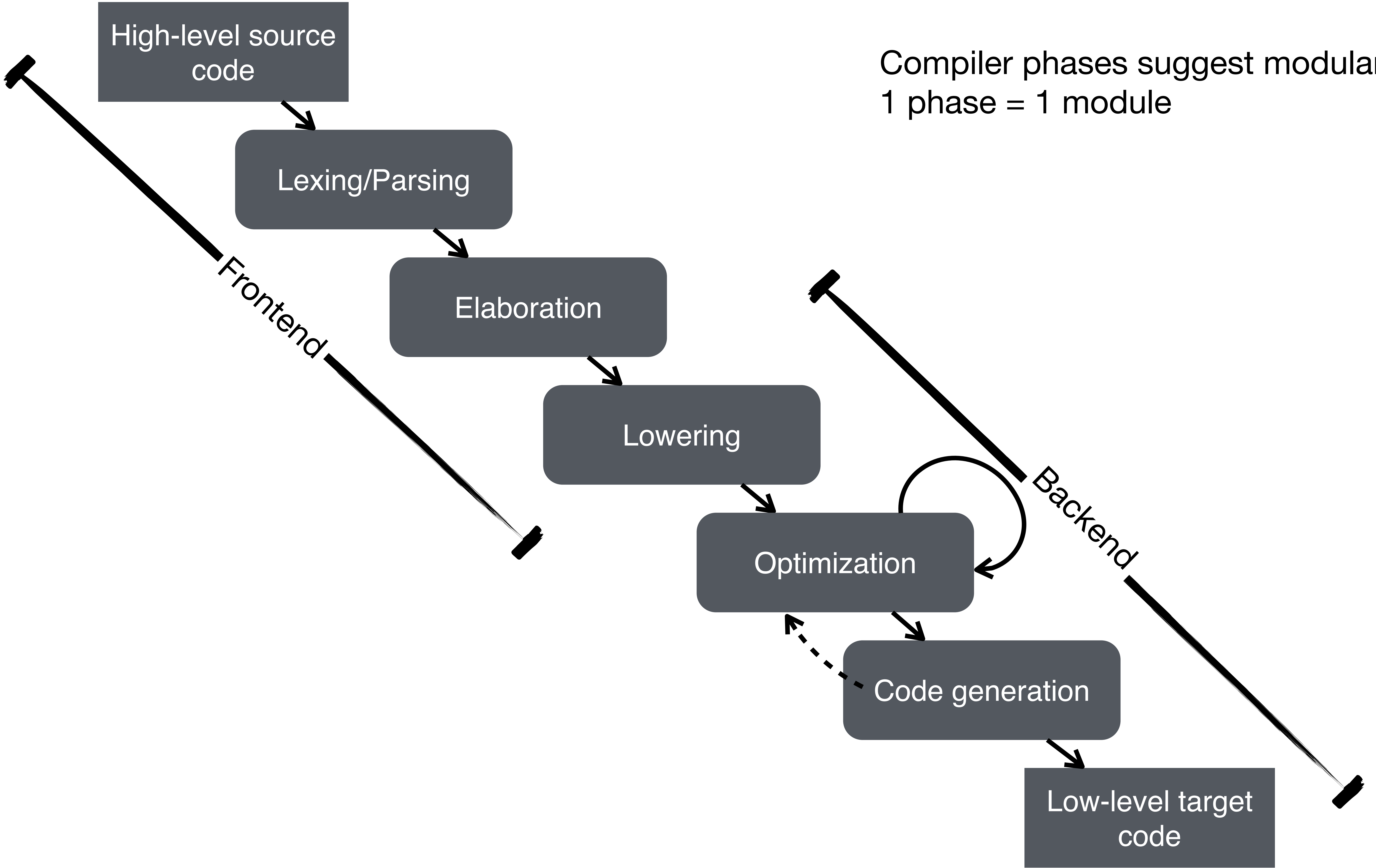
cf. compiler's folk theorem [citation needed ;)]

2024 CrowdStrike

A parsing error in a virtual machine that was pretending to be a device driver (which was cryptographically signed by Microsoft) running in a privileged mode in the OS kernel; the latter turn was apparently necessitated by an EU policy that prevented Microsoft to develop an appropriate kernel-level API; the whole thing driven by the necessity to quickly respond to ransomware, which is part of the CrowdStrike's business model.

Compiler phases

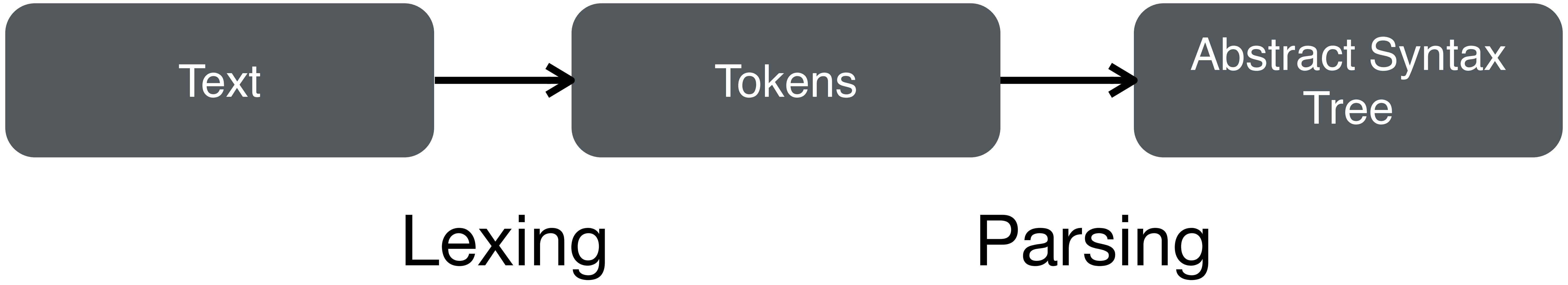
Basic phases of a compiler



Compiler phases suggest modular design
1 phase = 1 module

Front end

- Lexing & Parsing
  - From text to data structures
  - First two steps in processing from raw data to structured information
- Elegant application of CS theory
  - Regular expressions (finite state automata)
  - Context-free grammars (push-down automata)
- Established & streamlined tool support



Frontend: example

```
void printInt(i : int){
   output_string(int_to_string(i),get_stdout());
   output_string("\n",get_stdout());
}

int fact(n: int) {
    var x = n;
    var res = 1;
    while (x > 1) {
        res = res * x;
        x = x-1;
    };
    return res;
}

int main() {
    printInt(fact(5));
    printInt(fact(11));
    return 0;
}
```

Source program

```
VOID
IDENT<printInt>
LPAREN
IDENT<i>
COLON
INT
RPAREN
LBRACE
IDENT<output_string>
LPAREN
IDENT<int_to_string>
LPAREN
IDENT<i>
RPAREN
COMMA
IDENT<get_stdout>
LPAREN
RPAREN
RPAREN
SEMICOLON
IDENT<output_string>
LPAREN
STRING_LIT<
>
COMMA
IDENT<get_stdout>
LPAREN
RPAREN
RPAREN
SEMICOLON
RBRACE
INT
IDENT<fact>
LPAREN
IDENT<n>
COLON
INT
RPAREN
LBRACE
VAR
IDENT<x>
ASSIGN
IDENT<n>
SEMICOLON
VAR
IDENT<res>
ASSIGN
INT_LIT<1>
SEMICOLON
WHILE
LPAREN
IDENT<x>
GT
INT_LIT<1>
RPAREN
LBRACE
IDENT<res>
ASSIGN
IDENT<res>
MUL
IDENT<x>
SEMICOLON
IDENT<x>
ASSIGN
IDENT<x>
MINUS
INT_LIT<1>
SEMICOLON
RBRACE
SEMICOLON
RETURN
IDENT<res>
SEMICOLON
RBRACE
INT
IDENT<main>
LPAREN
RPAREN
LBRACE
IDENT<printInt>
LPAREN
IDENT<fact>
LPAREN
INT_LIT<5>
RPAREN
RPAREN
SEMICOLON
IDENT<printInt>
LPAREN
IDENT<fact>
LPAREN
INT_LIT<11>
RPAREN
RPAREN
SEMICOLON
RETURN
INT_LIT<0>
SEMICOLON
RBRACE
EOF
```

Stream of tokens

Frontend: example

```
VOID
IDENT<printInt>
LPAREN
IDENT<i>
COLON
INT
RPAREN
LBRACE
IDENT<output_string>
LPAREN
IDENT<int_to_string>
LPAREN
IDENT<i>
RPAREN
COMMA
IDENT<get_stdout>
LPAREN
RPAREN
RPAREN
SEMICOLON
IDENT<output_string>
LPAREN
STRING_LIT<
>
COMMA
IDENT<get_stdout>
LPAREN
RPAREN
RPAREN
SEMICOLON
RBRACE
INT
IDENT<fact>
LPAREN
IDENT<n>
COLON
INT
RPAREN
LBRACE
VAR
IDENT<x>
ASSIGN
IDENT<n>
SEMICOLON
VAR
IDENT<res>
ASSIGN
INT_LIT<1>
SEMICOLON
WHILE
LPAREN
IDENT<x>
GT
INT_LIT<1>
RPAREN
LBRACE
IDENT<res>
ASSIGN
IDENT<res>
MUL
IDENT<x>
SEMICOLON
IDENT<x>
ASSIGN
IDENT<x>
MINUS
INT_LIT<1>
SEMICOLON
```

Stream of tokens

```
Program
├─FunDecl
│ ├─RetType: Void
│ ├─FunName: printInt
│ ├─Params:
│ │ └─i : Int
│ └─Body
│   ├─ExprCmd: Call
│   │          ├─FunName: output_string
│   │          └─Args
│   │            ├─Call
│   │            │ ├─FunName: int_to_string
│   │            │ └─Args
│   │            │   └─Lval
│   │            │     └─Var(i)
│   │            └─Call
│   │              ├─FunName: get_stdout
│   │              └─Args
│   └─ExprCmd: Call
│              ├─FunName: output_string
│              └─Args
│                ├─StringLit("\n")
│                └─Call
│                  ├─FunName: get_stdout
│                  └─Args
├─FunDecl
│ ├─RetType: Int
│ ├─FunName: fact
│ ├─Params:
│ │ └─n : Int
│ └─Body
│   ├─DeclCmd: VarDecl
│   │          └─Declaration
│   │            ├─Ident: x
│   │            ├─Type:
│   │            └─Body: Lval
│   │                    └─Var(n)
│   ├─DeclCmd: VarDecl
│   │          └─Declaration
│   │            ├─Ident: res
│   │            ├─Type:
│   │            └─Body: IntLit(1)
│   ├─WhileCmd
│   │ ├─Cond: BinOp
│   │ │       ├─Lval
│   │ │       │ └─Var(x)
│   │ │       ├─Gt
│   │ │       └─IntLit(1)
│   │ └─Body: CompoundCmd
│   │         ├─ExprCmd: Assignment
│   │         │          ├─Var(res)
│   │         │          └─BinOp
│   │         │            ├─Lval
│   │         │            │ └─Var(res)
│   │         │            ├─Mul
│   │         │            └─Lval
│   │         │              └─Var(x)
│   │         └─ExprCmd: Assignment
│   │                    ├─Var(x)
│   │                    └─BinOp
│   │                      ├─Lval
│   │                      │ └─Var(x)
│   │                      ├─Minus
│   │                      └─IntLit(1)
│   ├─ExprCmd:
│   └─ReturnValCmd: Lval
│                   └─Var(res)
└─FunDecl
  ├─RetType: Int
  ├─FunName: main
  ├─Params:
  └─Body
    ├─ExprCmd: Call
    │          ├─FunName: printInt
    │          └─Args
    │            └─Call
    │              ├─FunName: fact
    │              └─Args
    │                └─IntLit(5)
    ├─ExprCmd: Call
    │          ├─FunName: printInt
    │          └─Args
    │            └─Call
    │              ├─FunName: fact
    │              └─Args
    │                └─IntLit(11)
    └─ReturnValCmd: IntLit(0)
```

Abstract Syntax Tree

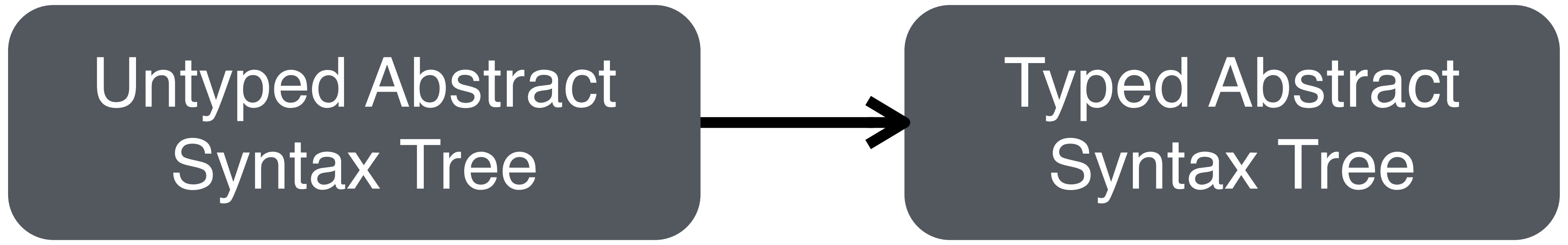
Semantic analysis

Resolving scope

Type checking

- Resolving variable types, modules, etc
- Check that operators and function calls are given the values of the right types
- Infer types for sub-expressions

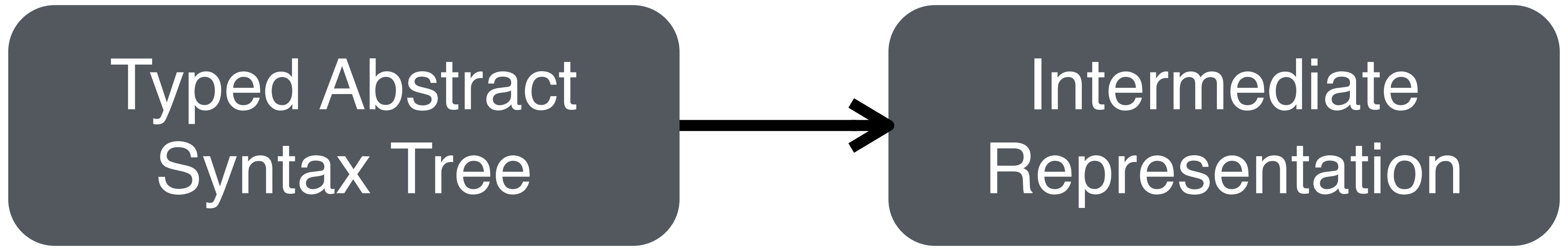
Most errors are reported to the user by the end of this phase



Lowering

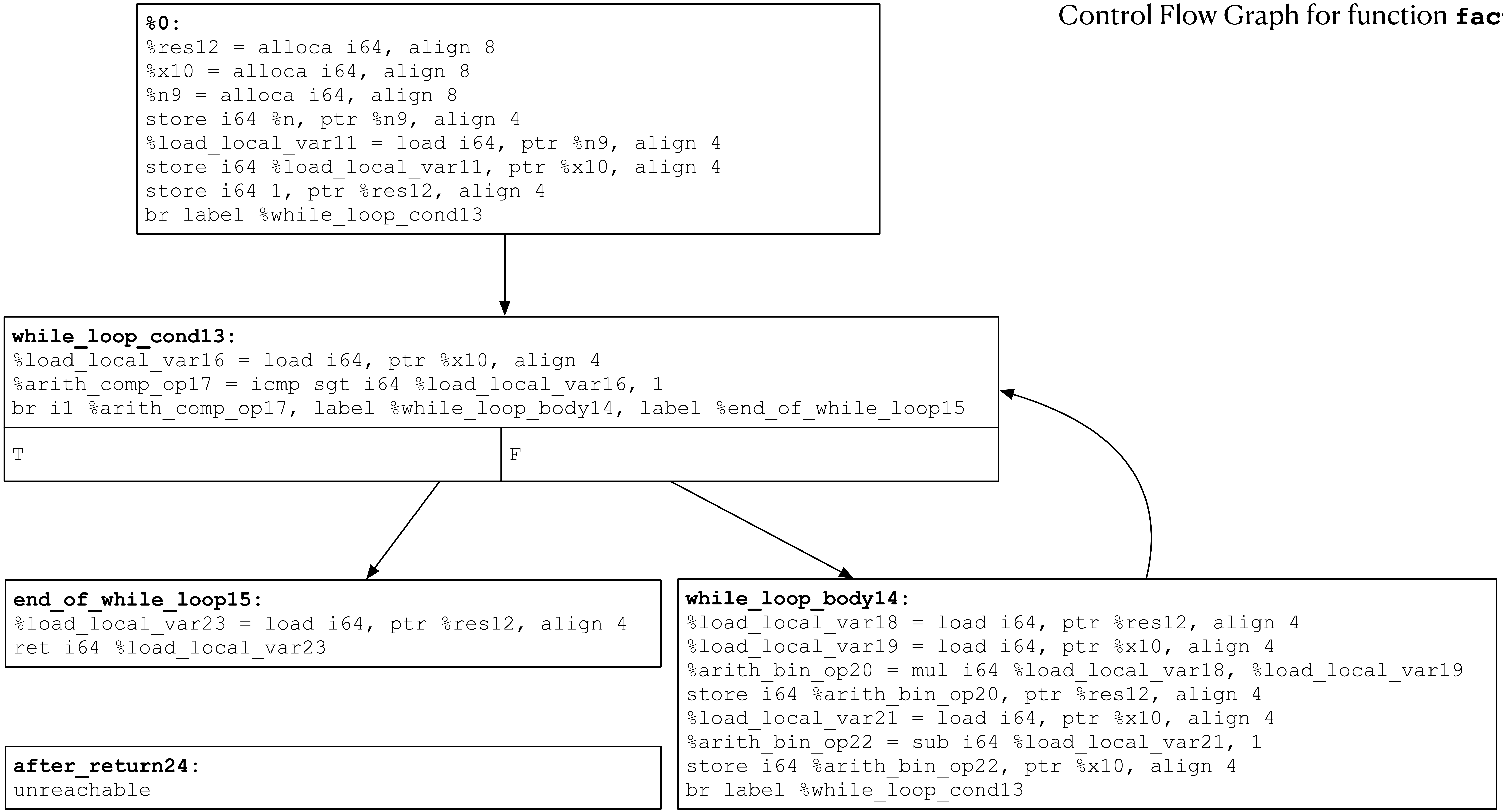
Translate high-level features into a small number of target-like constructs

- while, for - loops are all compiled to code using jumps
- embed array-bound checks, etc.

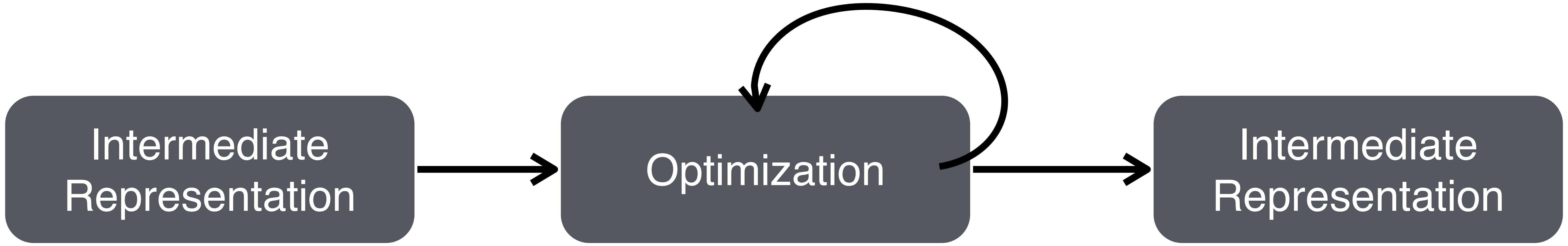


Intermediate representation

Control Flow Graph for function **fact**



Optimization



Detect expensive sequences of operations that can be rewritten into less expensive

Examples:

- constant folding: 2 + 2 → 4
- constant propagation: x = true; if x then A else B → if true then A else B
- dead code elimination if true then A else B → ~~if true then~~ A ~~else B~~ → A
- common subexpression elimination: x = a + b; y = a + b → x = a+b; y = x
- copy propagation: y = x; z = y + 1 → ~~y = x;~~ z = x + 1
- lifting invariant computations out of a loop
- loop parallelization
- inlining functions

Good news:

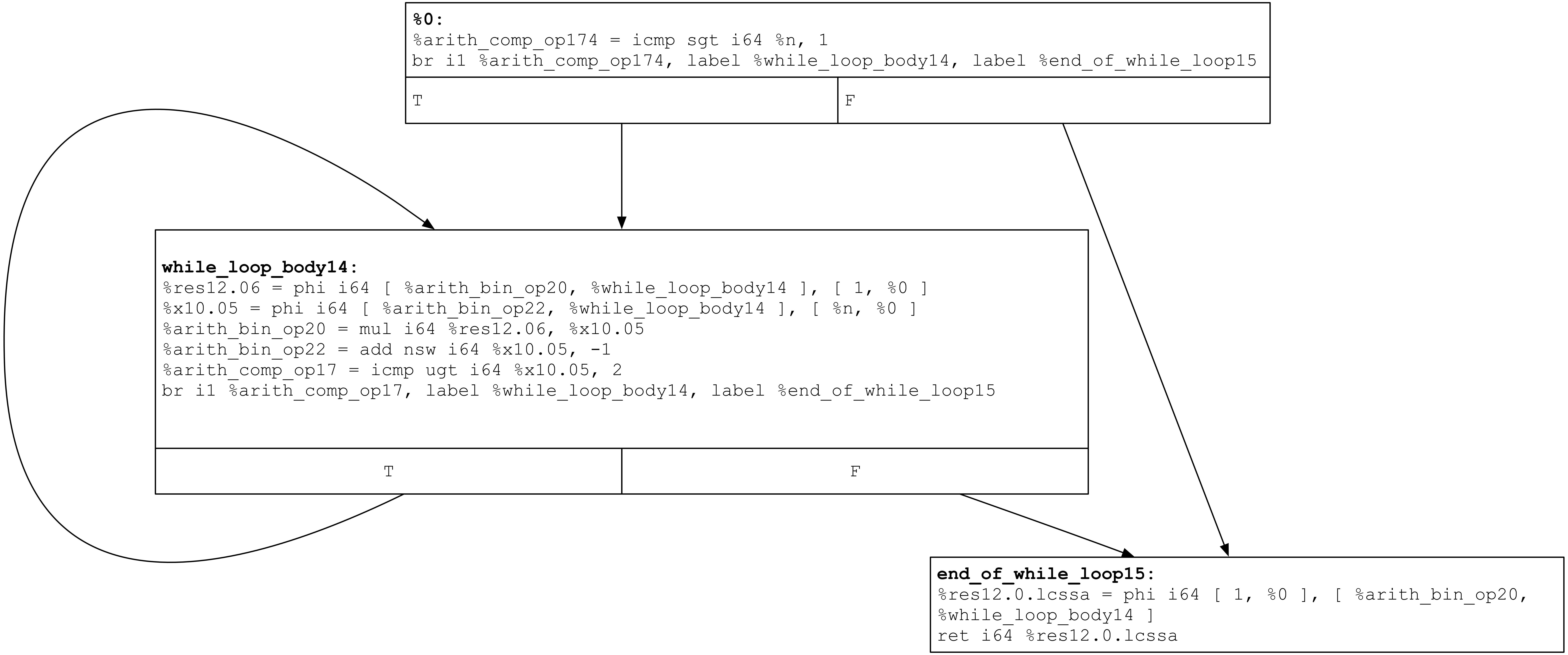
- Combining event the simplest of optimizations brings about substantial cumulative effect!
- Reduces the cost of the expensive niceties of the higher-level programming languages (objects, functions, exceptions)

Bad news

- Do not expect compiler to take your $O(n^2)$ program and turn it into $O(n \cdot \log n)$

Intermediate representation

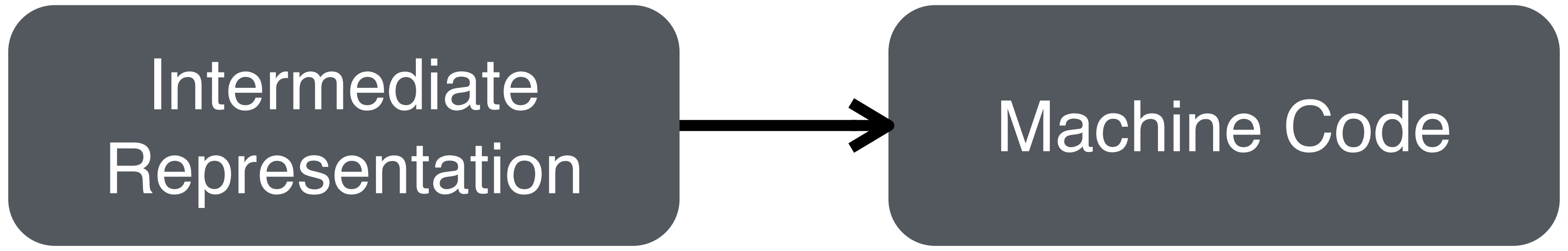
Control Flow Graph for function **`fact`** *after optimizations*



Code generation

Translate intermediate representation into target code

- Register assignment
- Instruction selection
- Instruction scheduling
- Machine-specific optimizations



Code generation

```
int fact(n: int) {
    var x = n;
    var res = 1;
    while (x > 1) {
        res = res * x;
        x = x-1;
    };
    return res;
}
```

```
## %bb.0:
    movl $1, %eax
    cmpq $2, %rdi
    jl   LBB1_3
## %bb.1:                                  ## %while_loop_body14.preheader
    movl $1, %eax
    .p2align 4, 0x90
LBB1_2:                                    ## %while_loop_body14
                                           ## =>This Inner Loop Header: Depth=1
    imulq    %rdi, %rax
    leaq -1(%rdi), %rcx
    cmpq $2, %rdi
    movq %rcx, %rdi
    ja   LBB1_2
LBB1_3:                                    ## %end_of_while_loop15
    retq
```

x86 Instructions

```
## %bb.0:
    movl $1, %eax
    cmpq $2, %rdi
    jl   LBB1_3
## %bb.1:
    movl $1, %eax
    .p2align 4, 0x90
LBB1_2:

    imulq    %rdi, %rax
    leaq -1(%rdi), %rcx
    cmpq $2, %rdi
    movq %rcx, %rdi
    ja   LBB1_2
LBB1_3:
    retq
```

Binary code

```
000023a0  e8 db fe ff ff 89 45 e0  48 8b 7d e8 e8 4f ff ff  |......E.H.}..O..|
000023b0  ff 89 45 dc 8b 7d e4 8b  75 e0 8b 55 dc e8 50 12  |..E..}..u..U..P.|
000023c0  00 00 89 45 d8 83 7d d8  ff 0f 85 0d 00 00 00 48  |...E..}........H|
000023d0  c7 45 f8 00 00 00 00 e9  5a 00 00 00 bf 30 00 00  |.E......Z....0..|
000023e0  00 e8 4a ec ff ff 48 89  45 d0 8b 4d d8 48 8b 45  |..J...H.E..M.H.E|
000023f0  d0 89 08 48 8b 4d f0 48  8b 45 d0 48 89 48 10 48  |...H.M.H.E.H.H.H|
00002400  8b 4d e8 48 8b 45 d0 48  89 48 08 48 8b 45 d0 c7  |.M.H.E.H.H.H.E..|
00002410  40 18 00 00 00 00 48 8b  45 d0 48 c7 40 20 00 00  |@.....H.E.H.@ ..|
00002420  00 00 48 8b 45 d0 48 c7  40 28 00 00 00 00 48 8b  |..H.E.H.@(....H.|
00002430  45 d0 48 89 45 f8 48 8b  45 f8 48 83 c4 30 5d c3  |E.H.E.H.E.H..0].|
00002440  55 48 89 e5 48 83 ec 10  48 89 7d f8 48 83 7d f8  |UH..H...H.}.H.}.|
00002450  00 0f 85 07 00 00 00 b0  00 e8 52 ef ff ff 48 8b  |..........R...H.|
00002460  45 f8 48 89 45 f0 48 8b  45 f0 83 78 18 03 0f 84  |E.H.E.H.E..x....|
00002470  24 00 00 00 48 8b 45 f0  83 78 18 04 0f 84 16 00  |$...H.E..x......|
00002480  00 00 48 8d 3d 1d 15 00  00 e8 72 11 00 00 bf 01  |..H.=.....r.....|
00002490  00 00 00 e8 02 11 00 00  48 8b 45 f0 48 8b 40 20  |........H.E.H.@ |
000024a0  48 83 c4 10 5d c3 66 2e  0f 1f 84 00 00 00 00 00  |H...].f.........|
000024b0  55 48 89 e5 48 83 ec 10  48 89 7d f8 48 83 7d f8  |UH..H...H.}.H.}.|
000024c0  00 0f 85 07 00 00 00 b0  00 e8 e2 ee ff ff 48 8b  |..............H.|
000024d0  45 f8 48 89 45 f0 48 8b  45 f0 83 78 18 03 0f 84  |E.H.E.H.E..x....|
000024e0  24 00 00 00 48 8b 45 f0  83 78 18 04 0f 84 16 00  |$...H.E..x......|
000024f0  00 00 48 8d 3d 03 15 00  00 e8 02 11 00 00 bf 01  |..H.=...........|
00002500  00 00 00 e8 92 10 00 00  48 8b 45 f0 48 8b 40 28  |........H.E.H.@(|
```

Phases of a real compiler (CompCert)

Properties of a good compiler

Generates correct code

Generates fast code

Conforms to the language specification

- neither super or subset, maximizes source portability

Supports arbitrary size input

- Do not assume that all input is human-generated
- Program-generated inputs stress compilers in different ways.

Good compilation speed

- Beware of sources of non-linearity
  - general-purpose CFG parsing is cubic
  - optimizations may be exponential in input size
- Language design should support separate compilation

Administrativia

Lectures are recorded on best-effort basis

Forum guidelines

- Generic questions — use the web forum
- Specific questions regarding infrastructure, OCaml
- Be nice :-)

Specific questions regarding the assignment: **ask your TAs**

Don't cheat/plagiarize

- Do not share your code or specifics of how you solve your assignment with students not in your group
- See AU rules on exam cheating and plagiarism

PeerWise (will be enabled later in the semester)

- Use this while studying the material to test yourself and others

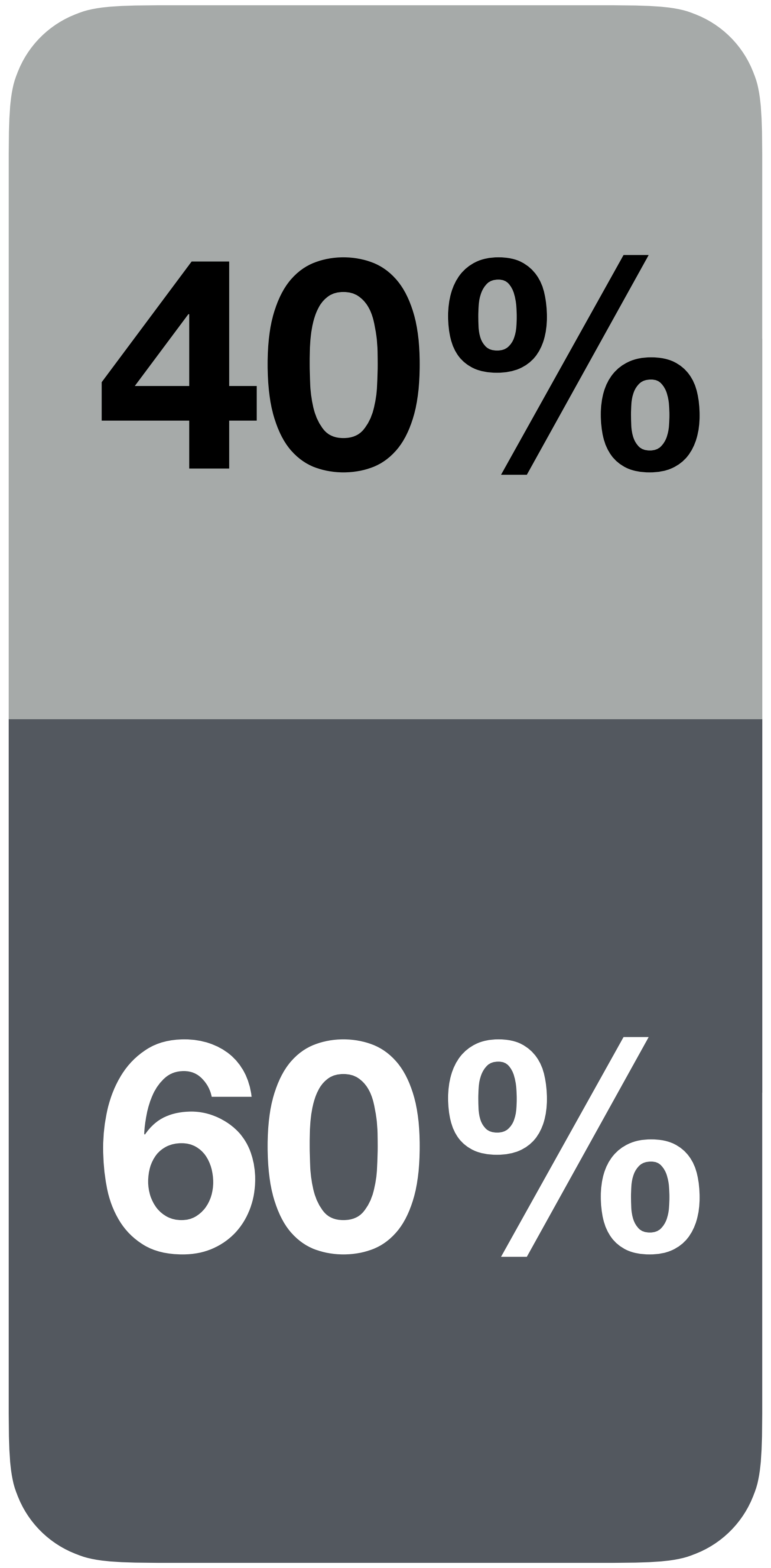
Assessment

9 assignments

Exam

Written, closed-book (no aids allowed), mostly multiple choice + write-in answers

The assignments are not a prerequisite for attending the exam



Final grade

Assignments

There are 9 programming assignments in the course

- Assignments 1 – 3 are individual
- Assignments 4 – 9 are in groups

Groups of size are to be registered on Brightspace

- The project workload is calibrated for 3-person groups

Implementation project

The language of no surprises

Compiler for the **Dolphin** programming language

- Own language created in 2023 year just for this course

Assignments

#	Description	Duration	I/G	Points
1	Arithmetic expressions, x86	1 w	Individual	20
2	Compiling arithmetic expressions to x86	2 w	Individual	50
3	Intermediate representation: LLVM	1 w	Individual	30
4	Dolphin subset 1: let bindings and conditionals to LLVM	2 w	Group	120
5	Dolphin subset 2: + loops and mutable vars to LLVM	1 w	Group	120
6	Dolphin subset 3: + frontend	2 w	Group	140
7	Dolphin subset 4: + functions	1	Group	80
8	Full Dolphin: + strings, records, arrays	2 w	Group	140
9	Full Dolphin: learning from past mistakes	1 w	Group	can regain 200 project points

The assignments sum up to 700 pts. You can claim maximum **600** pts for the 40% project part of the grade