Course Title:

Code Generation

NI(E)-GEN, Spring 2021

https://courses.fit.cvut.cz/NI-GEN

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

FIT

Lecture Title:

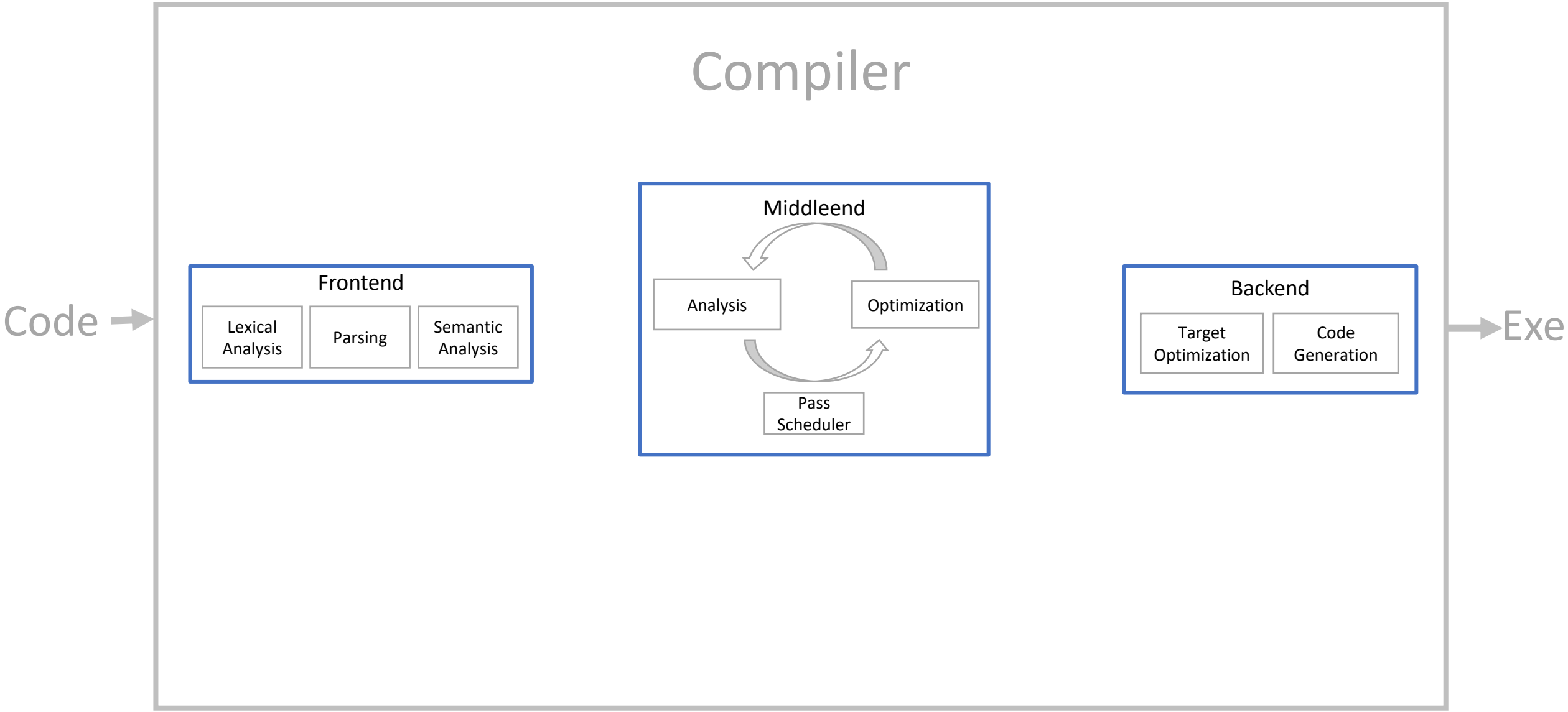
Code Generation

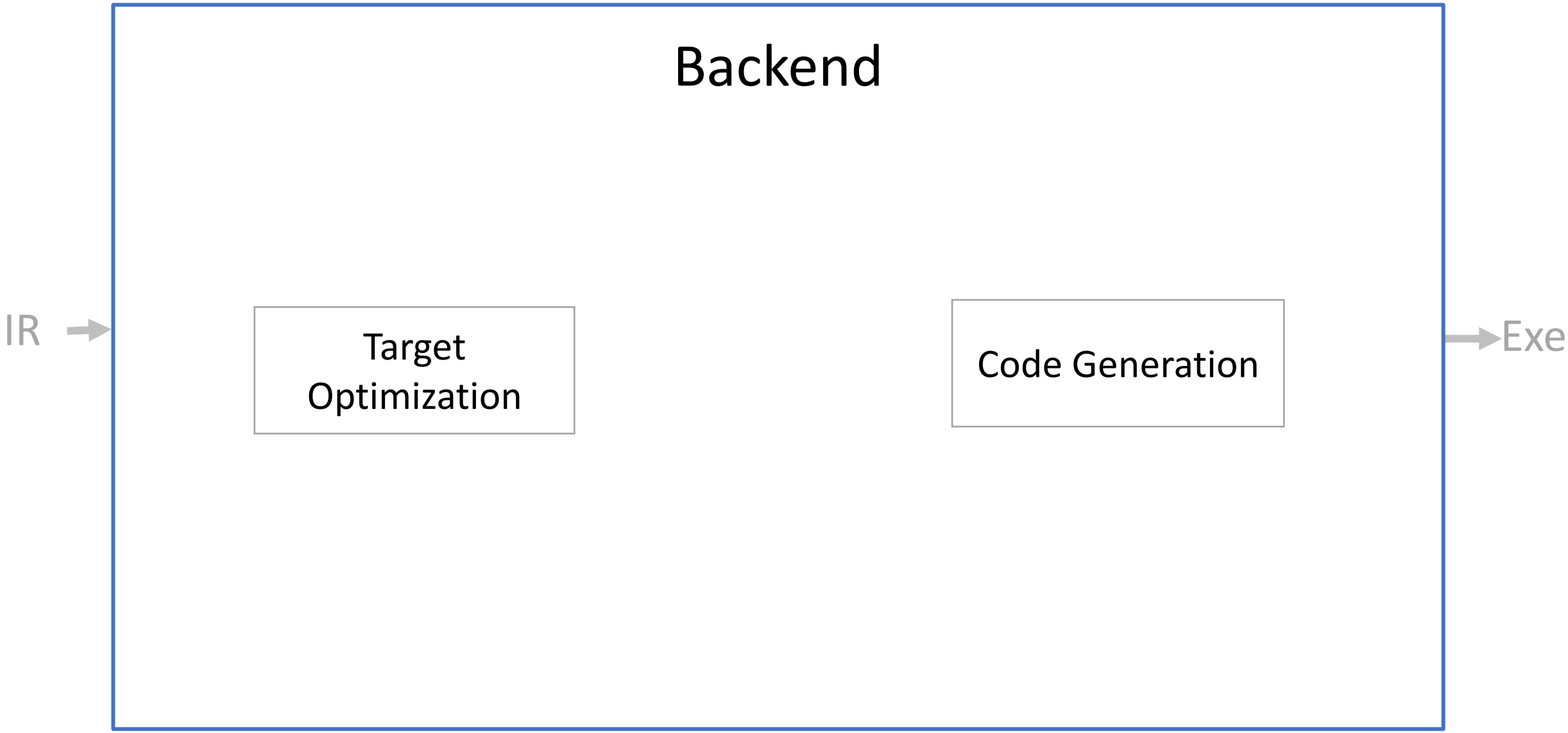
NI(E)-GEN, Spring 2021

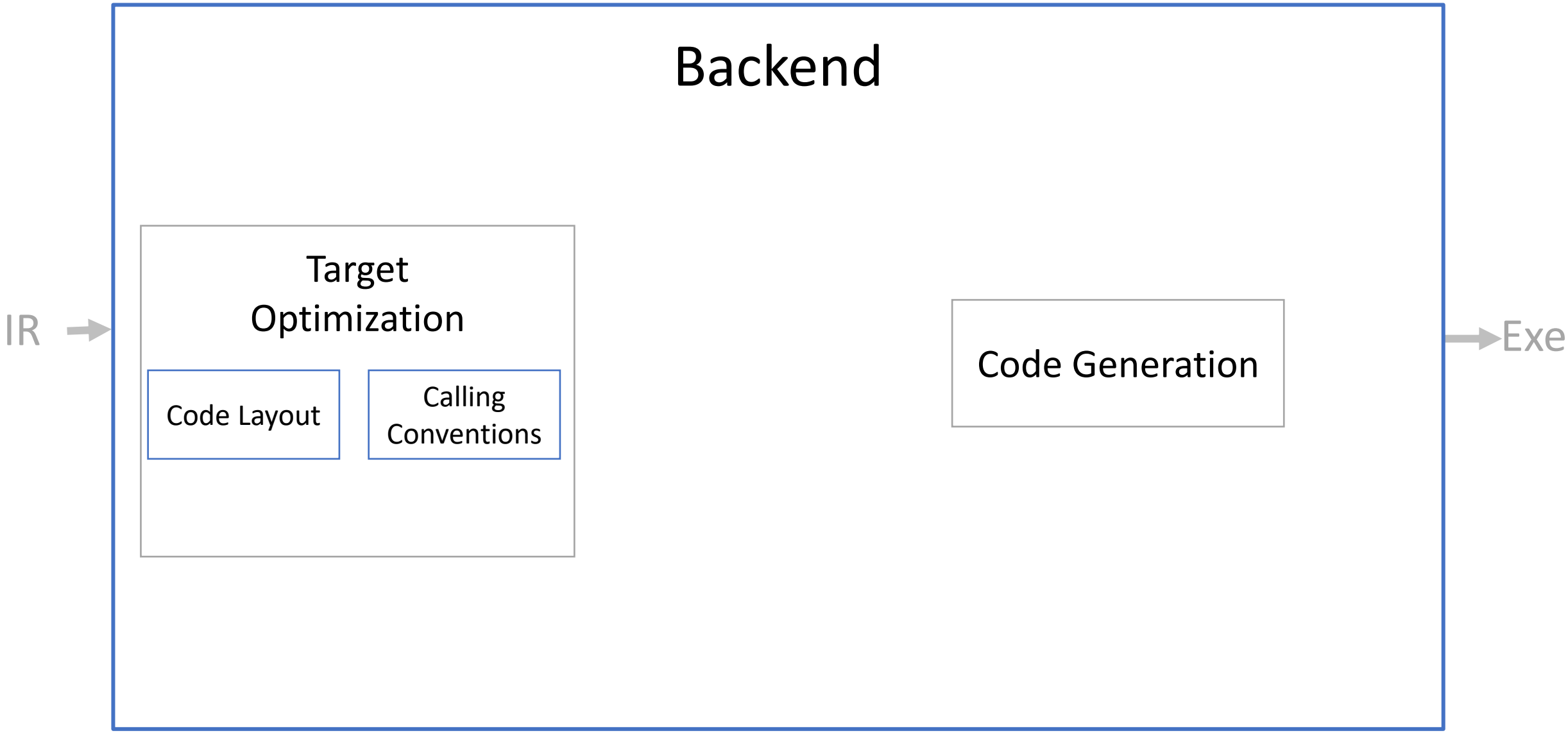
https://courses.fit.cvut.cz/NI-GEN

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

FIT


Compiler
Code
Frontend
Lexical Analysis
Parsing
Semantic Analysis
Middleend
Analysis
Optimization
Pass Scheduler
Backend
Target Optimization
Code Generation
Exe


Backend
IR
Target
Optimization
Code Generation
Exe


Backend
IR
Target
Optimization
Code Layout
Calling
Conventions
Code Generation
Exe

Code Layout

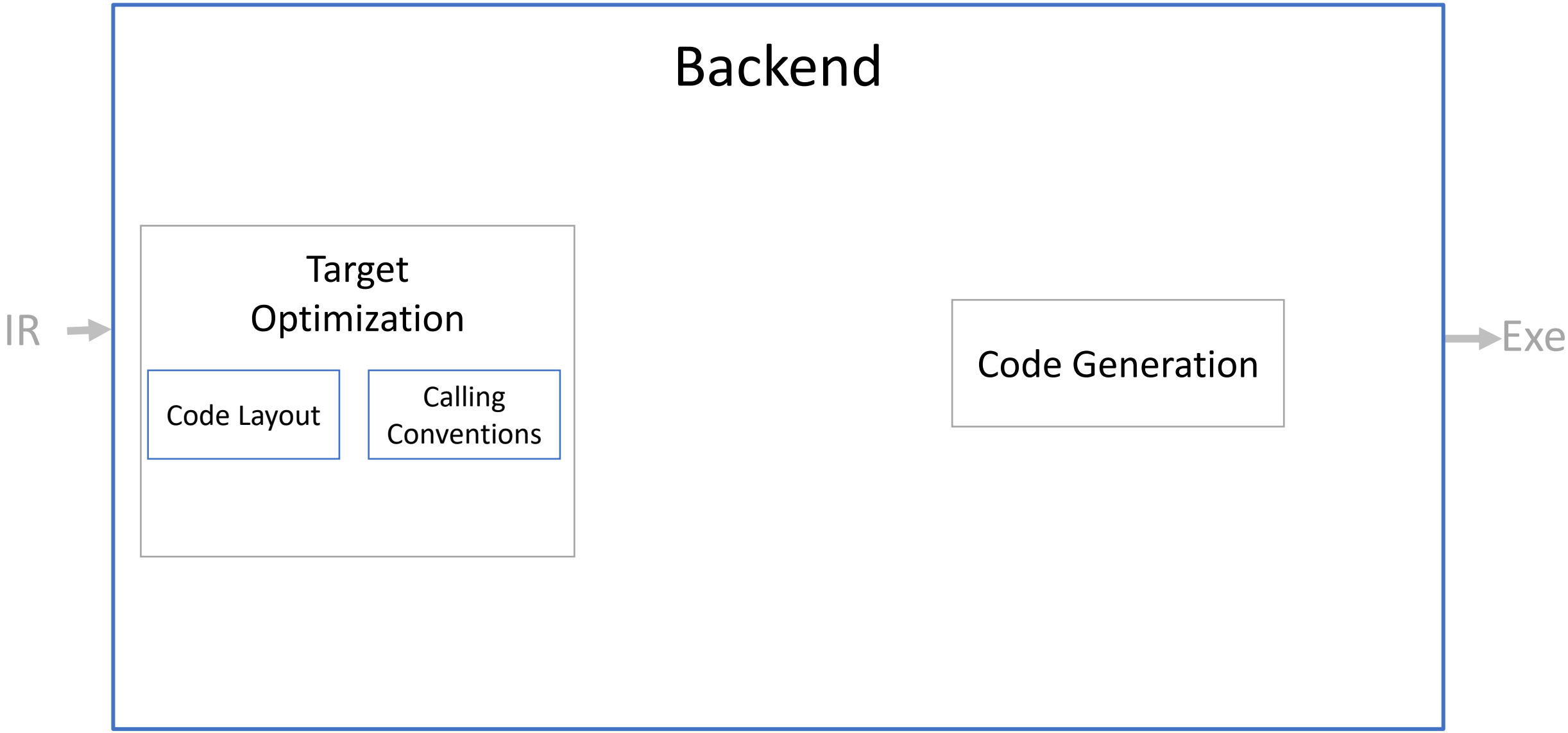
- determines the ordering of basic blocks in memory
- taking a branch is usually more expensive
  - longer execution time
  - worse code cache locality
- convert 2 target branches so that mostly taken branches don’t translate to jumps
  - not always possible
  - how to determine which target is more important?

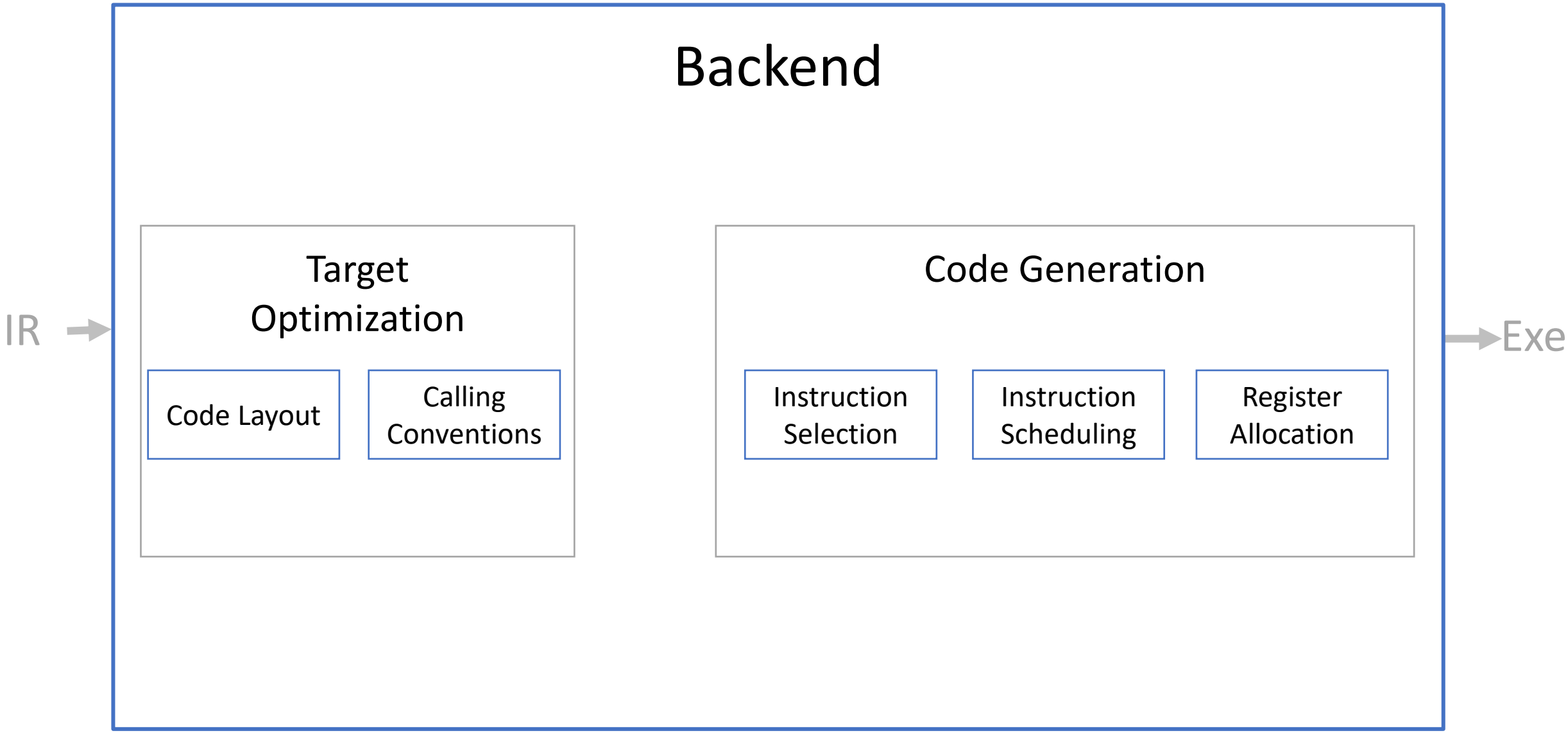
Calling Conventions

- calling convention for each function must be determined
- callers may need rewrite to provide arguments and read results properly (i.e. stack)
- callee must add prologue / epilogue code
  - stack & local variables bookkeeping
- local variables management

Local Variables

- on stack (addressable by statically known offsets from BP)
  - simple
  - slow (extra memory reads & writes)
- in register
  - uses machine registers (not that many of them)
  - not always possible
- usually determined by the optimizer and already reflected in some form in the IR
  - sometimes combination of both


Backend
IR
Target
Optimization
Code Layout
Calling
Conventions
Code Generation
Exe


Backend
IR
Target Optimization
Code Layout
Calling Conventions
Code Generation
Instruction Selection
Instruction Scheduling
Register Allocation
Exe

Code Generation

- traditionally consists of three main phases
  - instruction selection (determine which instructions will be used to execute the program's semantics)
  - instruction scheduling (reorder instructions to ease pressure on computer units)
  - register allocation (determine which machine registers will be used for what)

- presented separately

a hairy, interleaved, mess of code in practical reality…

Registers

- instruction selection almost never done in isolation
- the register allocator is responsible for providing new free registers
  - i.e. instead of r0, call nextRegister() which returns new free registers
- this may also insert extra code to keep old value if no new registers available
  - register spilling

Instruction Selection

Instruction Selection

- if IR is simple enough (such as our case), direct translation is possible
- for each instruction, generate its target counterpart (or counterparts)

```
a = b + 45;
```

a = b + 45;

r7 = **load** [b]

r8 = **load_imm** 45

r9 = **add** r7 r8

store [a] r9

a = b + 45;

r7 = **load** [b]

r8 = **load_imm** 45

r9 = **add** r7 r8

store [a] r9

mov r1, [b]

mov r2, 45

add r1, r2

mov [a], r1

a = b + c + d + e + f + g + h + i + j + 45;

Say we only have 8 registers...

a = b + c + d + e + f + g + h + i + j + 45;

r0 = **load** [b]

r1 = **load** [c]

r2 = **load** [d]

r3 = **load** [e]

r4 = **load** [f]

r5 = **load** [g]

r6 = **load** [h]

r7 = **load** [i]

r8 = **load** [j]

r9 = **load_imm** 45

r10 = **add** r8 r9

store [a] r10

a = b + c + d + e + f + g + h + i + j + 45;

mov r0, [b]
mov r1, [c]
mov r2, [d]
mov r3, [e]
mov r4, [f]
mov r5, [g]
mov r6, [h]
mov r7, [i]
mov ??, [j]

mov ??, 45
…

a = b + c + d + e + f + g + h + i + j + 45;

mov r0, [b]

mov r1, [c]

mov r2, [d]

mov r3, [e]

mov r4, [f]

mov r5, [g]

mov r6, [h]

mov r7, [i]

mov ??, [j]

mov ??, 45

…

Instruction Selection

- if IR is simple enough (such as our case), direct translation is possible
- for each instruction, generate its target counterpart (or counterparts)
- keep local variables in memory
  - lowers register pressure
  - expensive

Naïve Instruction Selection

- if IR is simple enough (such as our case), direct translation is possible
- for each instruction, generate its target counterpart (or counterparts)
- keep local variables in memory
  - lowers register pressure
  - expensive

a = b + 45;

r7 = **load** [b]

r8 = **load_imm** 45

r9 = **add** r7 r8

store [a] r9

mov r1, [b]

mov r2, [45]

add r1, r2

mov [a], r1

a = b + 45;

r7 = **load** [b]
r8 = **load_imm** 45
r9 = **add** r7 r8
store [a] r9

mov r1, [b]
mov r2, 45
add r1, r2
mov [a], r1

mov r1, [b]
add r1, 45
mov [a], r1

a = b + c + d + e + f + g + h + i + j + 45;

mov r0, [b]

mov r1, [c]

mov r2, [d]

mov r3, [e]

mov r4, [f]

mov r5, [g]

mov r6, [h]

mov r7, [i]

mov ??, [j]

mov ??, 45

…

a = b + c + d + e + f + g + h + i + j + 45;

```
mov r0, [b]
add r0, [c]
add r0, [d]
add r0, [e]
add r0, [f]
add r0, [g]
add r0, [h]
add r0, [i]
add r0, 45
```

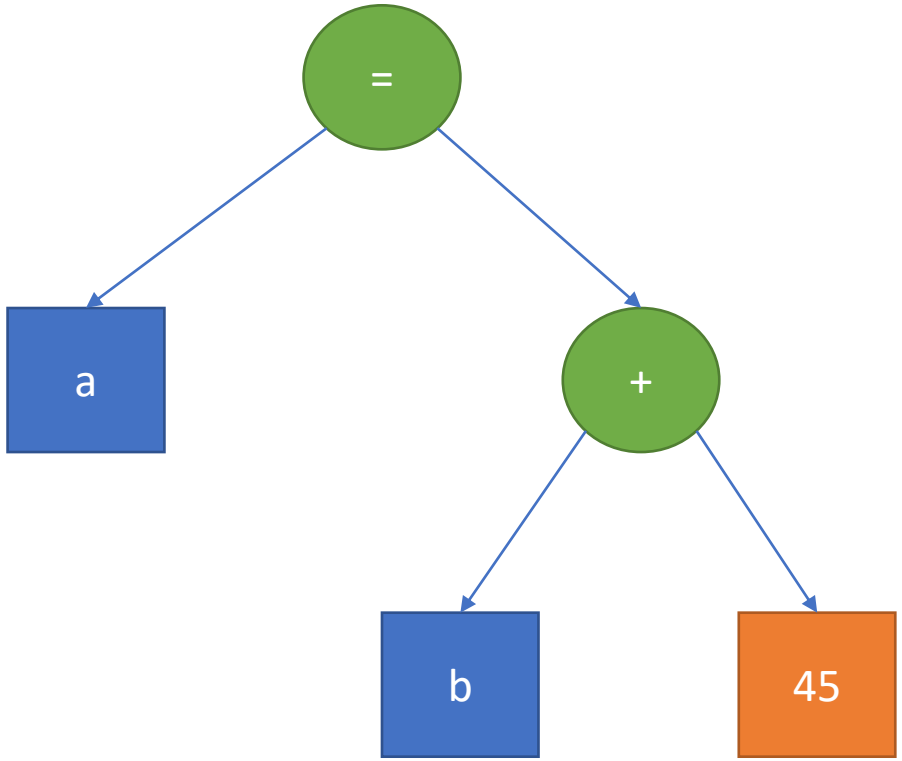
```
mov [a], r0
```

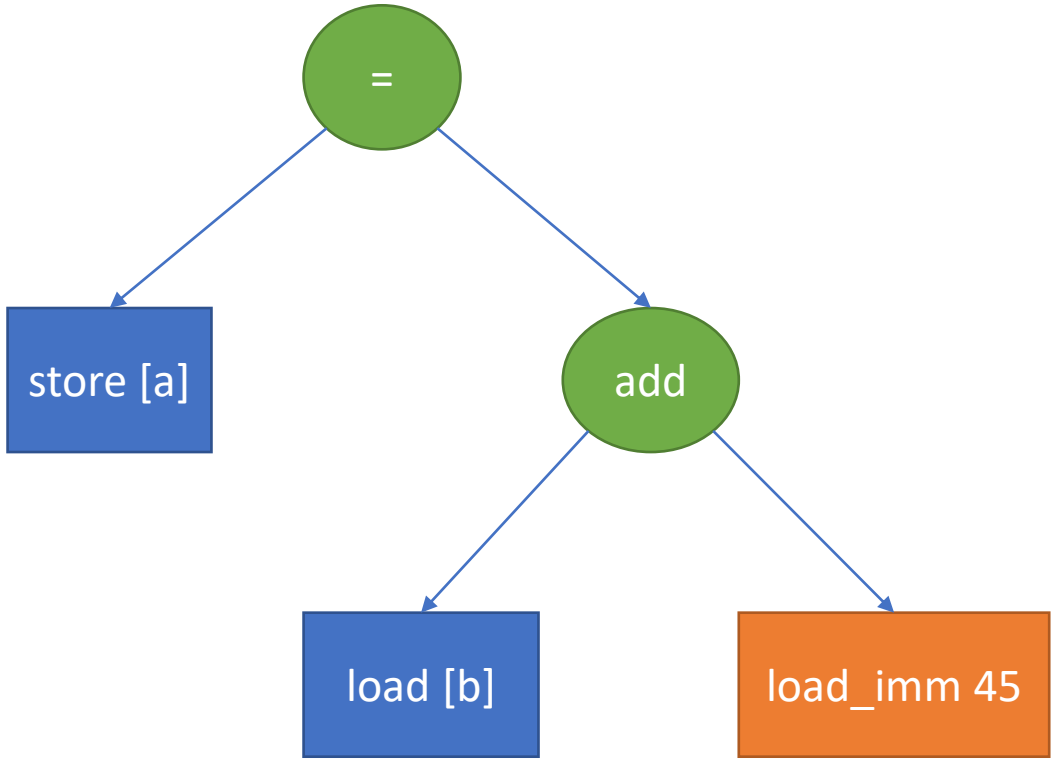
Instruction Selection II

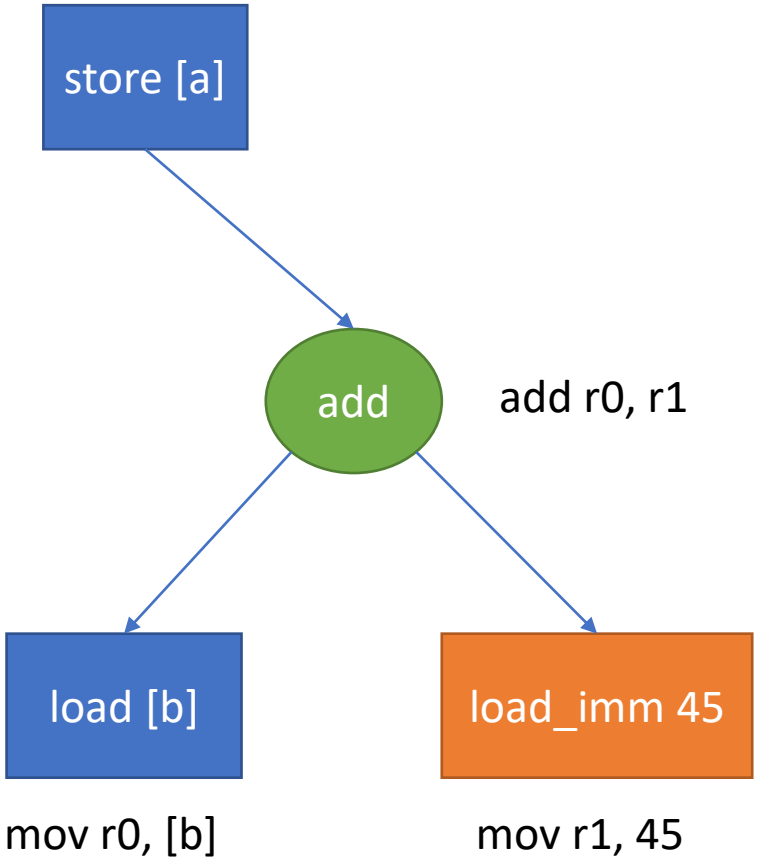
- naïve implementation possible, but suboptimal
- same operation can be expressed by many different instructions
  - much more so for CISC machines
  - or when changing archictecture (stack IR to register machine, etc.)
- different instructions can be beneficial in different settings
  - speed vs size vs power consumption

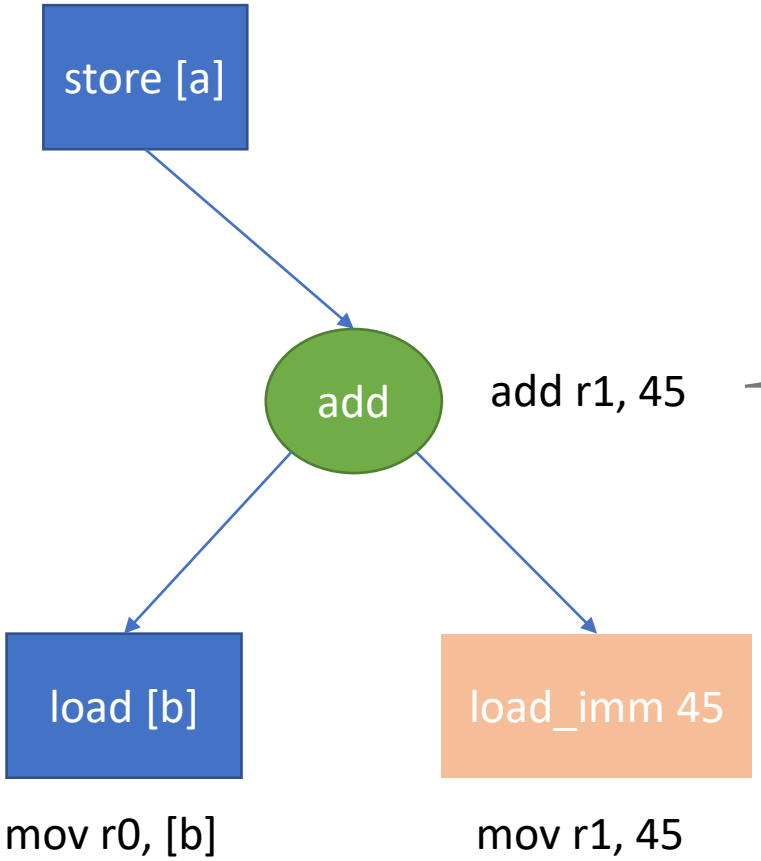
Tree Walk

- bottom-up walk of the IR tree
- rewrite rules for each node


=
a
+
b
45

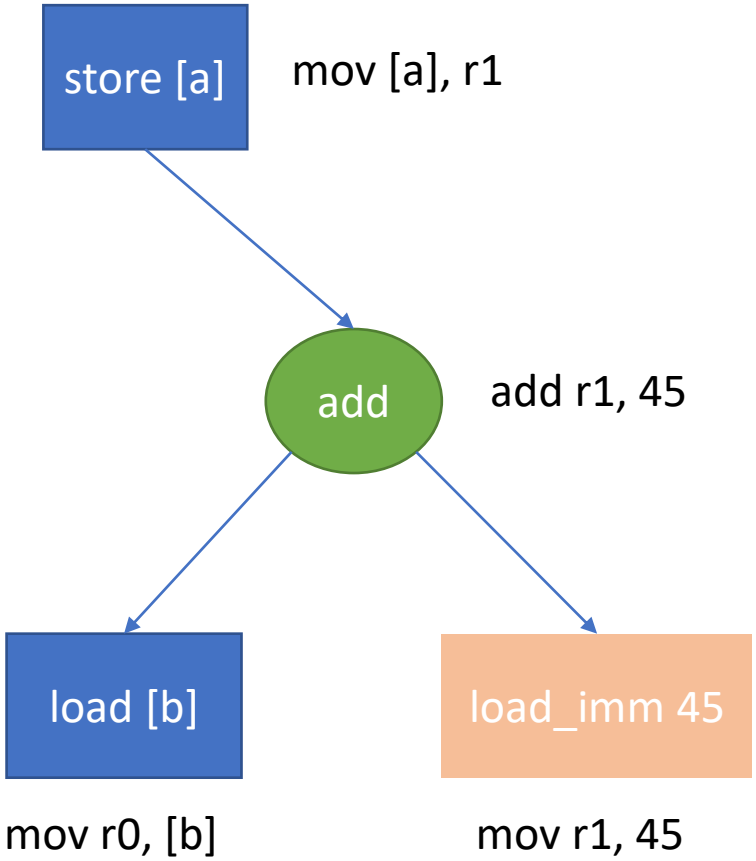

=
store [a]
add
load [b]
load_imm 45

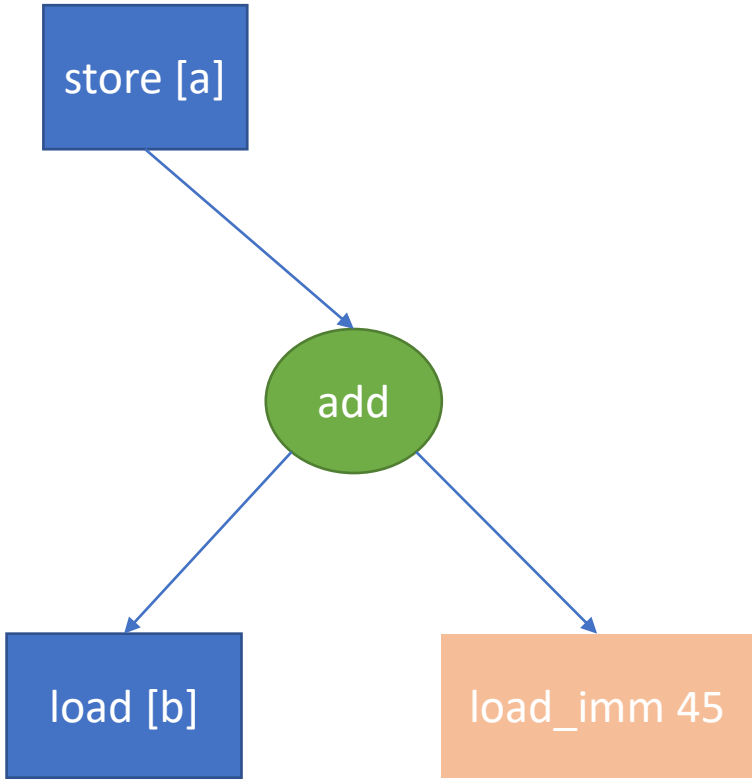

store [a]
add
add r0, r1
load [b]
load_imm 45
mov r0, [b]
mov r1, 45



Rule : if rhs or lhs is constant, rewrite as add immediate and do not use rhs

a hack, not really bottom up any more


store [a]
mov [a], r1
add
add r1, 45
load [b]
load_imm 45
mov r0, [b]
mov r1, 45



mov r1, 45

The generated code:

mov r0, [b]

add r1, 45

mov [a], r1

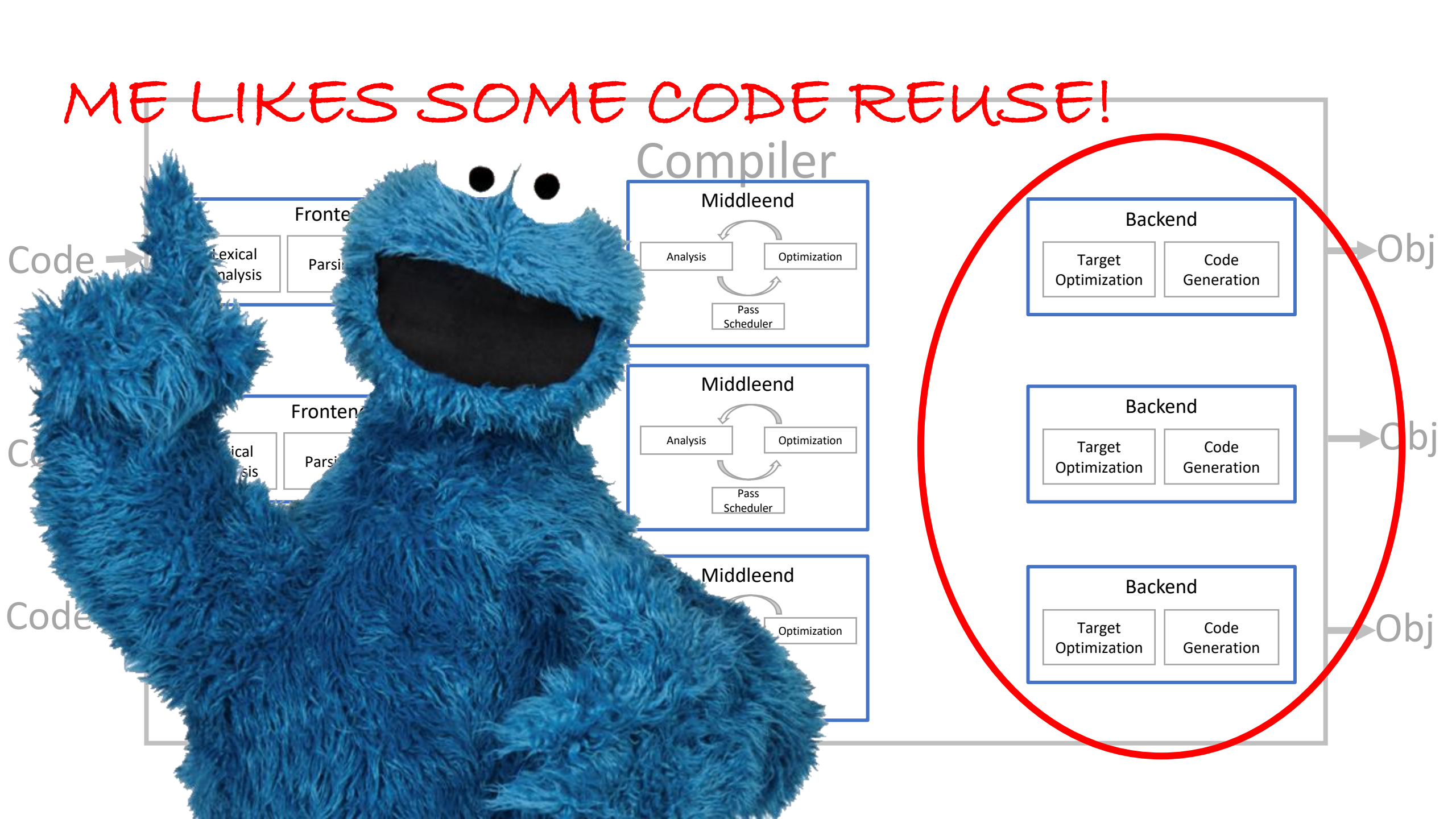
Tree Walk

- bottom-up walk of the IR tree
- rewrite rules for each node
  - quality depends on the complexity of the rules
- only explores local optimizations
  - cf. optimizing AST-based IR
- should be followed by target code optimizer to remove obvious inefficiencies
  - double loads / extra stores, etc.

Tree Walk

- often extra nodes (dereferencing, etc.) are preserved in the AST
- this allows better code matching opportunities
  - at the expense of costlier tree rewrite
  - and more complex rewrite rules
- rewrite rules can be ambiguous and have associated cost with them
  - different costs can be used for different goals (speed, size, etc.)

ME LIKES SOME CODE REUSE!



Tree Tiling

- generalization of the tree-walk method
- can be expressed in pattern tables for each architecture and share the actual implementation code
- tree must encode operations and inputs (register, memory, immediate)
  - immediate sizes, register constraints, etc.

Rewriting Rules

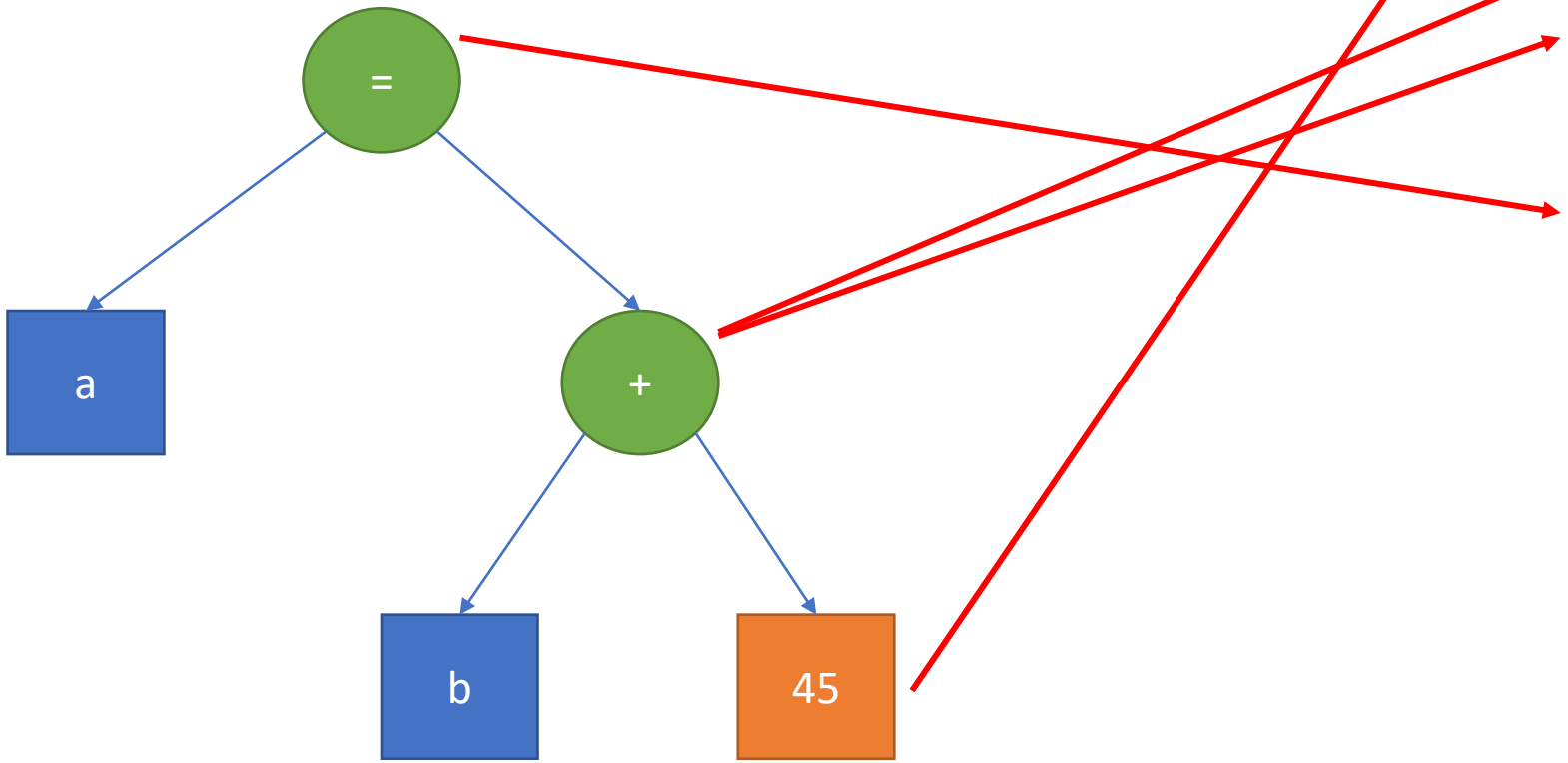
AST	Target
IMM	mov REGNEW, IMM
* REG1	mov REGNEW, [REG1]
* IMM	mov REGNEW, [IMM]
* + (REG1, IMM)	mov REGNEW, [REG1 + IMM]
+ (REG1, REG2)	add REG1, REG2
+ (REG1, IMM)	add REG1, IMM
+ (REG1, * REG2)	add REG1, [REG2]
+ (REG1, * + (REG2, IMM))	add REG1, [REG2 + IMM]
= (* REG1, REG2)	mov [REG1], REG2

and many, many more...

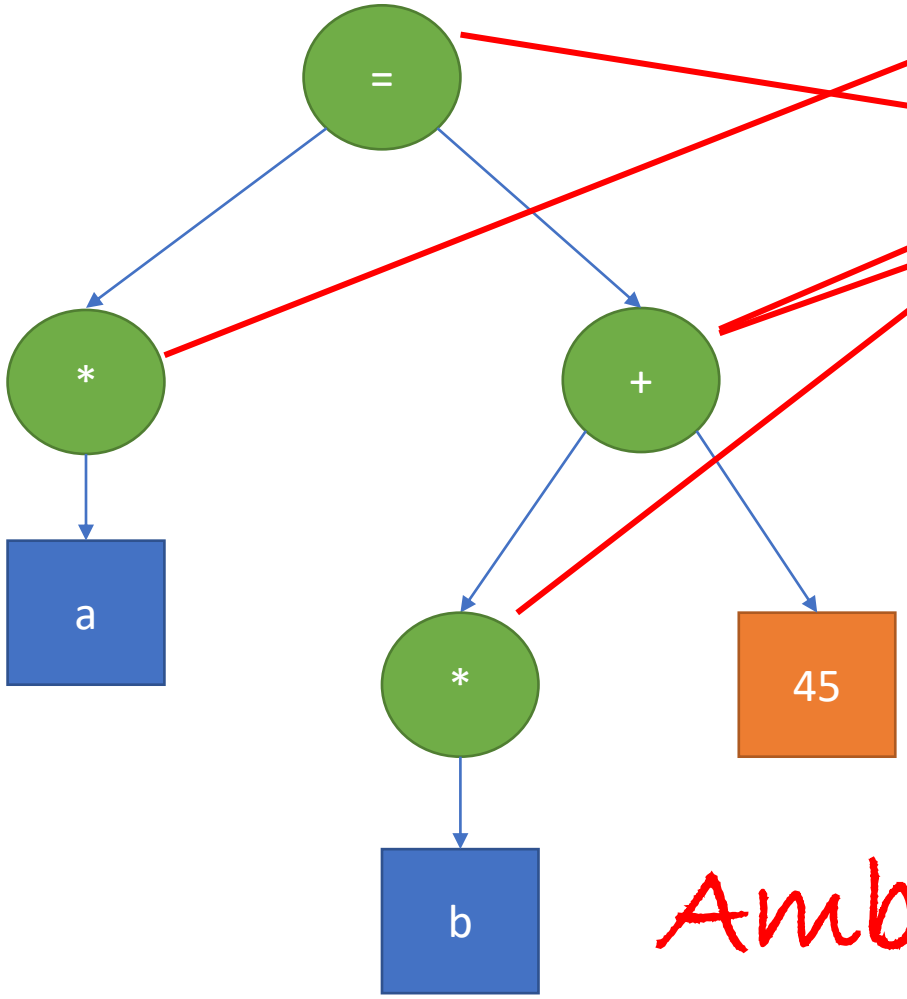
Rewriting Rules are actually a grammar

AST	Target
REG := IMM	mov REGNEW, IMM
REG := * REG1	mov REGNEW, [REG1]
REG := * IMM	mov REGNEW, [IMM]
REG := * + (REG1, IMM)	mov REGNEW, [REG1 + IMM]
REG := + (REG1, REG2)	add REG1, REG2
REG := + (REG1, IMM)	add REG1, IMM
REG := + (REG1, * REG2)	add REG1, [REG2]
REG := + (REG1, * + (REG2, IMM))	add REG1, [REG2 + IMM]
ASSIGN := = (* REG1, REG2)	mov [REG1], REG2

and many, many more...

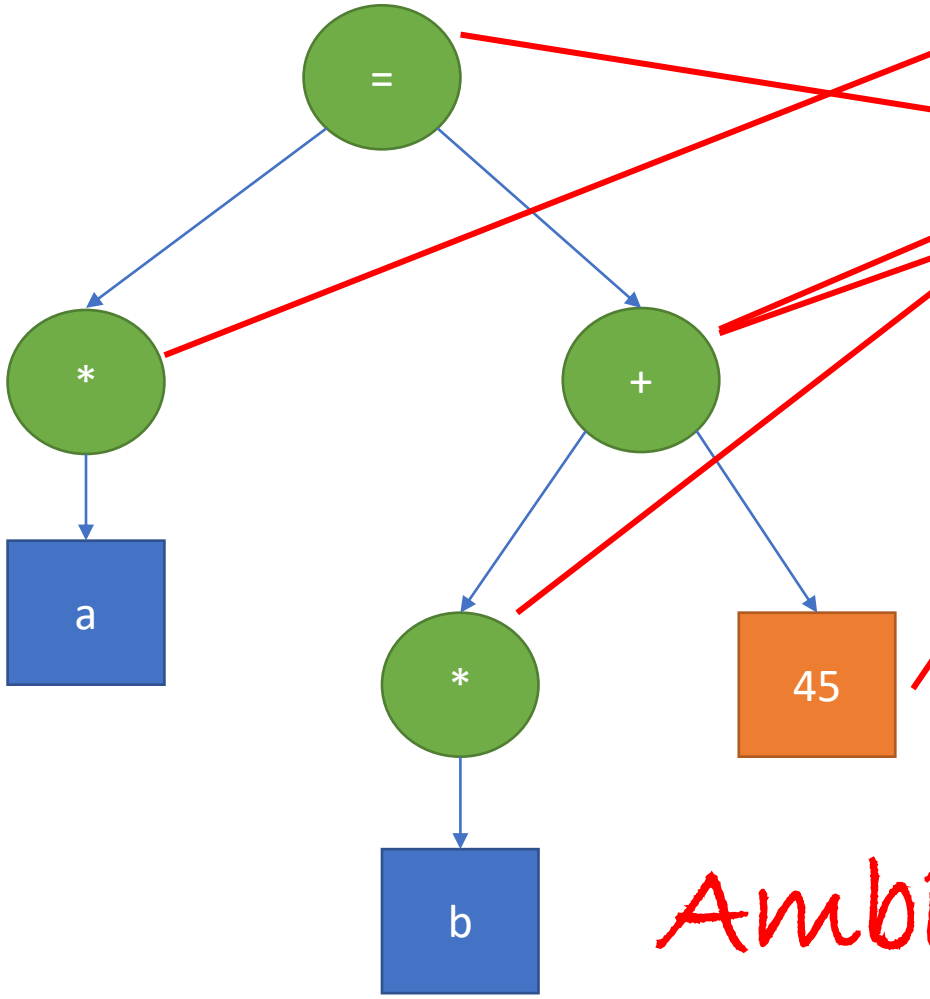


AST	Target
IMM	mov REGNEW, IMM
* REG1	mov REGNEW, [REG1]
* IMM	mov REGNEW, [IMM]
* + (REG1, IMM)	mov REGNEW, [REG1 + IMM]
+ (REG1, REG2)	add REG1, REG2
+ (REG1, IMM)	add REG1, IMM
+ (REG1, * REG2)	add REG1, [REG2]
+ (REG1, * + (REG2, IMM))	add REG1, [REG2 + IMM]
= (* REG1, REG2)	mov [REG1], REG2



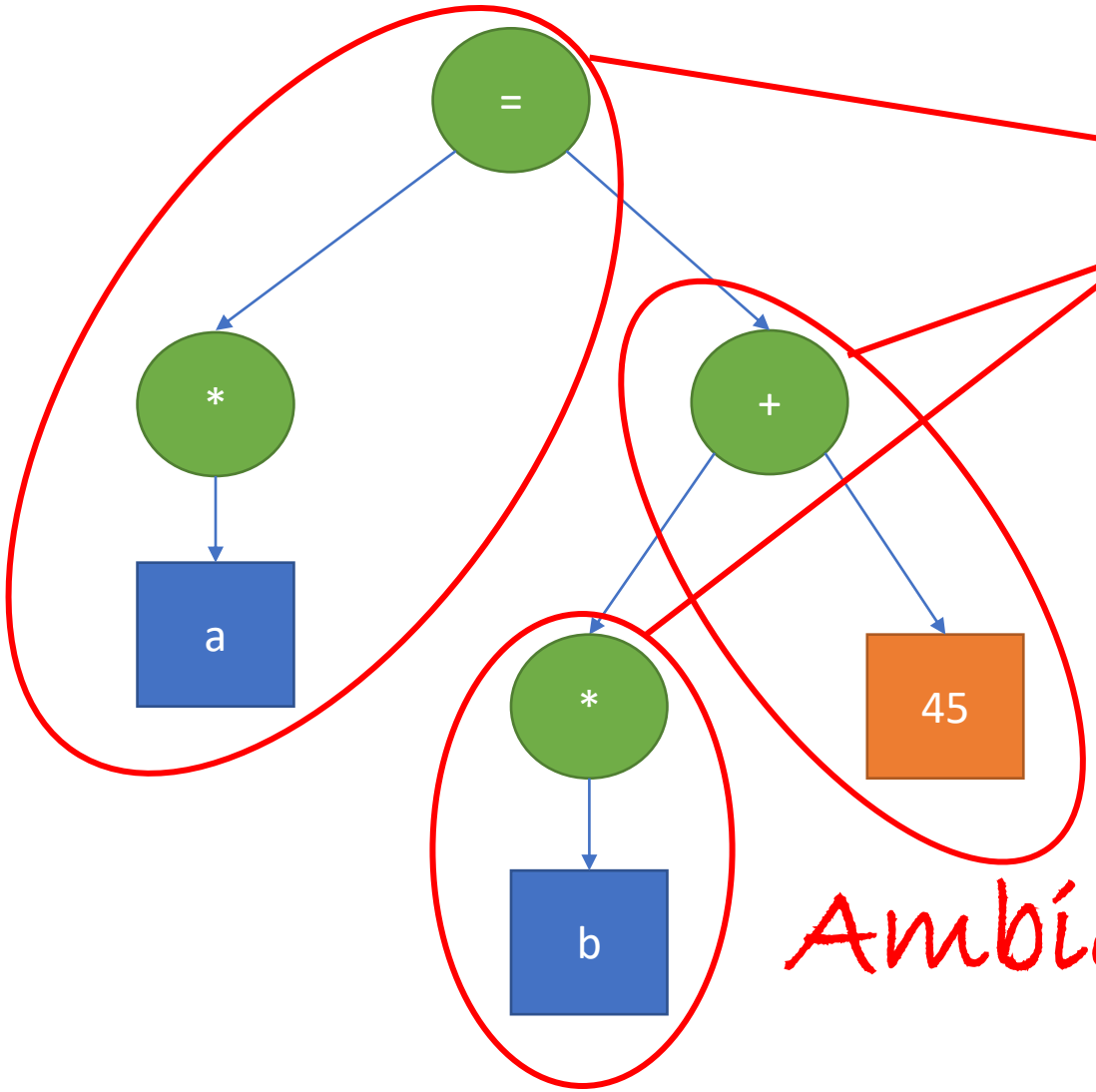
AST	Target
IMM	mov REGNEW, IMM
* REG1	mov REGNEW, [REG1]
* IMM	mov REGNEW, [IMM]
* + (REG1, IMM)	mov REGNEW, [REG1 + IMM]
+ (REG1, REG2)	add REG1, REG2
+ (REG1, IMM)	add REG1, IMM
+ (REG1, * REG2)	add REG1, [REG2]
+ (REG1, * + (REG2, IMM))	add REG1, [REG2 + IMM]
= (* REG1, REG2)	mov [REG1], REG2

Ambiguity



AST	Target	$
IMM	mov REGNEW, IMM	1
* REG1	mov REGNEW, [REG1]	1
* IMM	mov REGNEW, [IMM]	1
* + (REG1, IMM)	mov REGNEW, [REG1 + IMM]	1
+ (REG1, REG2)	add REG1, REG2	1
+ (REG1, IMM)	add REG1, IMM	1
+ (REG1, * REG2)	add REG1, [REG2]	1
+ (REG1, * + (REG2, IMM))	add REG1, [REG2 + IMM]	1
= (* REG1, REG2)	mov [REG1], REG2	1

Ambiguity



AST	Target	$
IMM	mov REGNEW, IMM	1
* REG1	mov REGNEW, [REG1]	1
* IMM	mov REGNEW, [IMM]	1
* + (REG1, IMM)	mov REGNEW, [REG1 + IMM]	1
+ (REG1, REG2)	add REG1, REG2	1
+ (REG1, IMM)	add REG1, IMM	1
+ (REG1, * REG2)	add REG1, [REG2]	1
+ (REG1, * + (REG2, IMM))	add REG1, [REG2 + IMM]	1
= (* REG1, REG2)	mov [REG1], REG2	1

Ambiguity

mov r1, [b]
add r1, 45
mov [a], r1

Finding All Tilings

- to simplify, assume at most 2 subtrees per node

- assume that each rule contains at most one operation

Ouch:

REG := + (REG1, * + (REG2, IMM))

Now illegal:(

Finding All Tilings

- to simplify, assume at most 2 subtrees per node

- assume that each rule contains at most one operation
  - simple rewrite, replace the extra operation with special symbol only appearing once, assign cost of 0 to it

REG := + (REG1, * + (REG2, IMM))
becomes:
REG := + (REG1, REG_TMP_1) [$1]
REG_TMP_1 := * REG_TMP_2 [$0]
REG_TMP_2 := + (REG, IMM) [$0]

Finding All Tilings

- to simplify, assume at most 2 subtrees per node
- assume that each rule contains at most one operation
  - simple rewrite, replace the extra operation with special symbol only appearing once, assign cost of 0 to it
- task is to find for each node a list of rules that can be used to tile it
  - the rule also tells us about its lhs and rhs subtrees, and thus are sufficient to reconstruct the tiling

For Each Binary Tile:

labels(**n**) = empty set

Start empty

tile(left(**n**))

tile(right(**n**))

Calculate labels for subtrees

for each rule **r** that matches **n**'s operation:

if left(**r**) ∈ labels(left(**n**)) and right(**r**) ∈ labels(right(**n**))

labels(**n**) += **r**

Check all possible own rules, see if the lhs and rhs nonterminals of the rule match labels for the lhs and rhs subtrees respectively. If they do add the rule to own labels.

For Each Unary Tile:

labels(**n**) = empty set

Start empty

tile(arg(**n**))

calculate label for the operand subtree

for each rule **r** that matches **n**'s operation

if arg(**r**) ∈ labels(arg(**n**))

labels(**n**) += **r**

Check all possible own rules, see if the operand nonterminals of the rule match labels for the operand subtree. If they do add the rule to own labels.

For Each Leaf Tile:

labels(**n**) = empty set

Start empty

for each rule **r** that matches **n**'s operation

labels(**n**) += **r**

There is nothing to check for leaves:)

Finding All Tilings

- naïve implementations will be slow
- repeated rule searches can be optimized in tables
  - these tend to get very large, but are really sparse
- rules can be pruned
  - each node can only keep rules that provide the cheapest tiling

Peephole Optimizations

- a simple, yet powerful concept for many backend optimizations
- idea: by observing and modifying a relatively small sliding window of instructions (peephole), the compiler can discover **local** optimizations
- used not just for instruction selection
  - expression simplification
  - dead store, extra load removal
  - …

a = b + 45;

(A) The trivial generator:

```
mov r1, [b]
mov r2, 45
add r1, r2
mov [a], r1
```

(B) The optimal code:

```
mov r1, [b]
add r1, 45
mov [a], r1
```

The usual problem: How to get from A to B?

a = b + 45;

```
mov r1, [b]
mov r2, 45
add r1, r2
mov [a], r1
```

```
mov r1, [b]
add r1, 45
mov [a], r1
```

a = b + 45;

```
mov r1, [b]
mov r2, 45
add r1, r2
mov [a], r1
```

```
mov r1, [b]
add r1, 45
mov [a], r1
```

Available rewrite:

```
mov R1, IMM
add R2, R1
```

-> add R2, IMM

a = b + 45;

```
mov r1, [b]
add r1, 45
mov [a], r1
```

```
mov r1, [b]
add r1, 45
mov [a], r1
```

Yeah!

The Devil Lies In The Detail

- simple peepholers offer very limited optimization possibilities
  - can only match what it sees
  - context only as large as the peephole
- modern implementations consist of three steps
  - expansion
  - simplification
  - matching

Expansion

- takes each IR operation and expands it into a low-level set of operations
  - these capture every side-effect of the operation
  - setting values, flags, etc.

- exposes “hidden” instructions
  - reading local variable via BP,
  - getting function arguments
  - reading values from memory (unless explicit in IR)

Simplification

- looks at the contents of sliding window in the low level IR
- at each step looks through a database to find simpler rewrites within
- if found, rewrites and tries again on rewritten code
- if none found, moves the window

Matching

- rewrites the simplified low level IR to target assembly
- usually simple matching of N operations

The Devil Lies In The Detail II

- must know the liveness of variables and remove dead ones
  - removing dead variable declaration removes instruction and changes the contents of the peephole
- the devil lies in the detail III
  - this is hard with control flow, code generation on single basic block basis wastes optimization opportunities
- what should be the peephole size?
  - more is not always better
- the window does not have to be sequential instructions, but can use DDG for more context as well

On The Other Side

- well-made peepholer can be powerful
- can be easily retargeted
- can easily add new features (adding rewrite rules)
- the simplifier provides distinct benefits compared to tree rewriting
  - and plays better with the other backend tasks

Improving Instruction Selection

Improving Instruction Selection

- vital for compiler performance
- new architecture versions often change instruction parameters
- really complex for advanced instruction sets
  - even RISC can get really complex nowadays
  - vectorization, special modes, etc.

There are bugs too: (

Learning IS Patterns

- both tree tiling and peepholer benefit from many useful rewrite patterns
- getting these is lengthy and involved process, but can be precalculated
  - try to write these for the tiny86 backend
- for small peephole sizes, even exhaustive search of all possible patterns is feasible
  - law of diminishing returns

Superoptimizers

- a traditional compiler is unlikely to produce the optimal code for any given problem
- a supercompiler attempts to find the true optimal solution

Superoptimizer -- A Look at the Smallest Program

Henry Massalin

Department of Computer Science
Columbia University
New York, NY 10027

Abstract

Given an instruction set, the superoptimizer finds the shortest program to compute a function. Startling programs have been generated, many of them engaging in convoluted bit-fiddling bearing little resemblance to the source programs which defined the functions. The key idea in the superoptimizer is a probabilistic test that makes exhaustive searches practical for programs of useful size. The search space is defined by the processor's instruction set, which may include the whole set, but it is typically restricted to a subset. By constraining the instructions and observing the effect on the output program, one can gain insight into the design of instruction sets. In addition, superoptimized programs may be used by peephole optimizers to improve the quality of generated code, or by assembly language programmers to improve manually written code.

pare the superoptimizer with related work. The conclusion in section 6 is followed by a list of interesting minimal programs in appendix I.

2. An Interesting Example

We begin with an example to show what superoptimized code looks like. The instruction set used here, as in most of the paper, is Motorola's 68020 instruction set. Our example is the *signum* function, defined by the following program:

```
signum(x)
int     x;
{
        if(x > 0)       return 1;
        else if(x < 0)  return -1;
        else            return 0;
}
```

This function compiles to 9 instructions occupying 18 bytes of

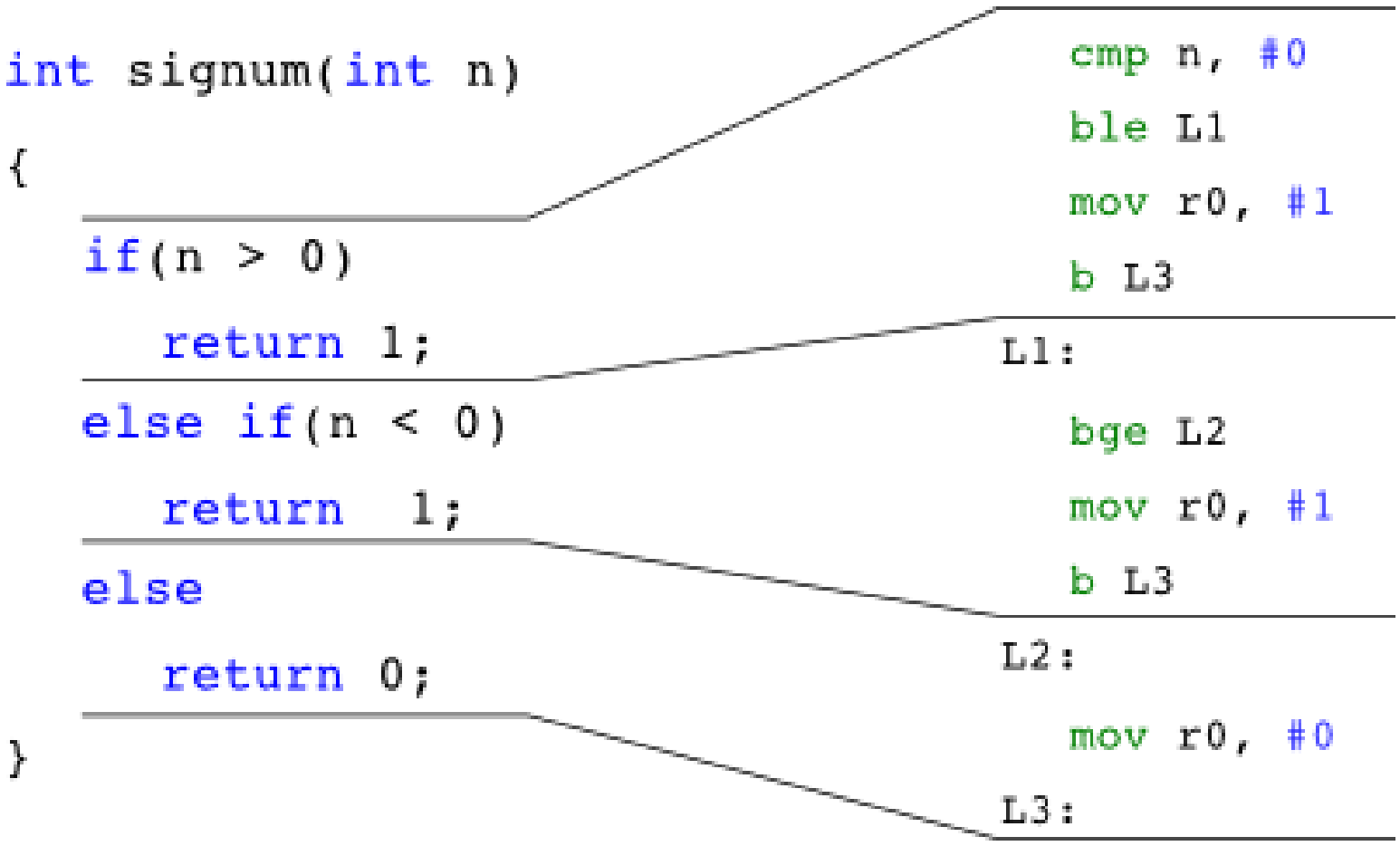
Superoptimizers

- a traditional compiler is unlikely to produce the optimal code for any given problem
- a supercompiler attempts to find the true optimal solution

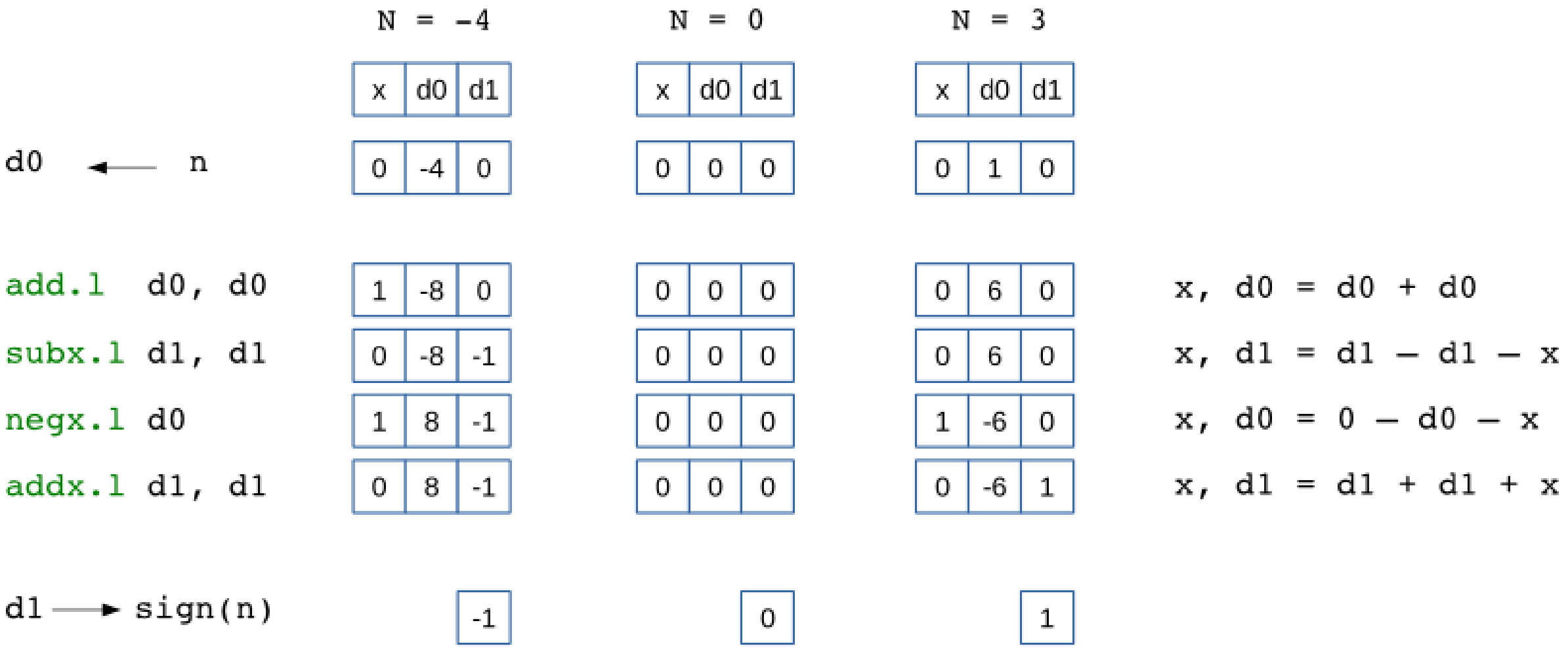
- does so by exhaustive brute force search of all possible programs of given sizes
  - scales not well
- or SAT solvers
  - scales better, but programs are vast!

```
int signum(int n)
{
   if(n > 0)
      return 1;
   else if(n < 0)
      return  1;
   else
      return 0;
}
```

http://superoptimization.org/wiki/Superoptimizing_Compilers



http://superoptimization.org/wiki/Superoptimizing_Compilers

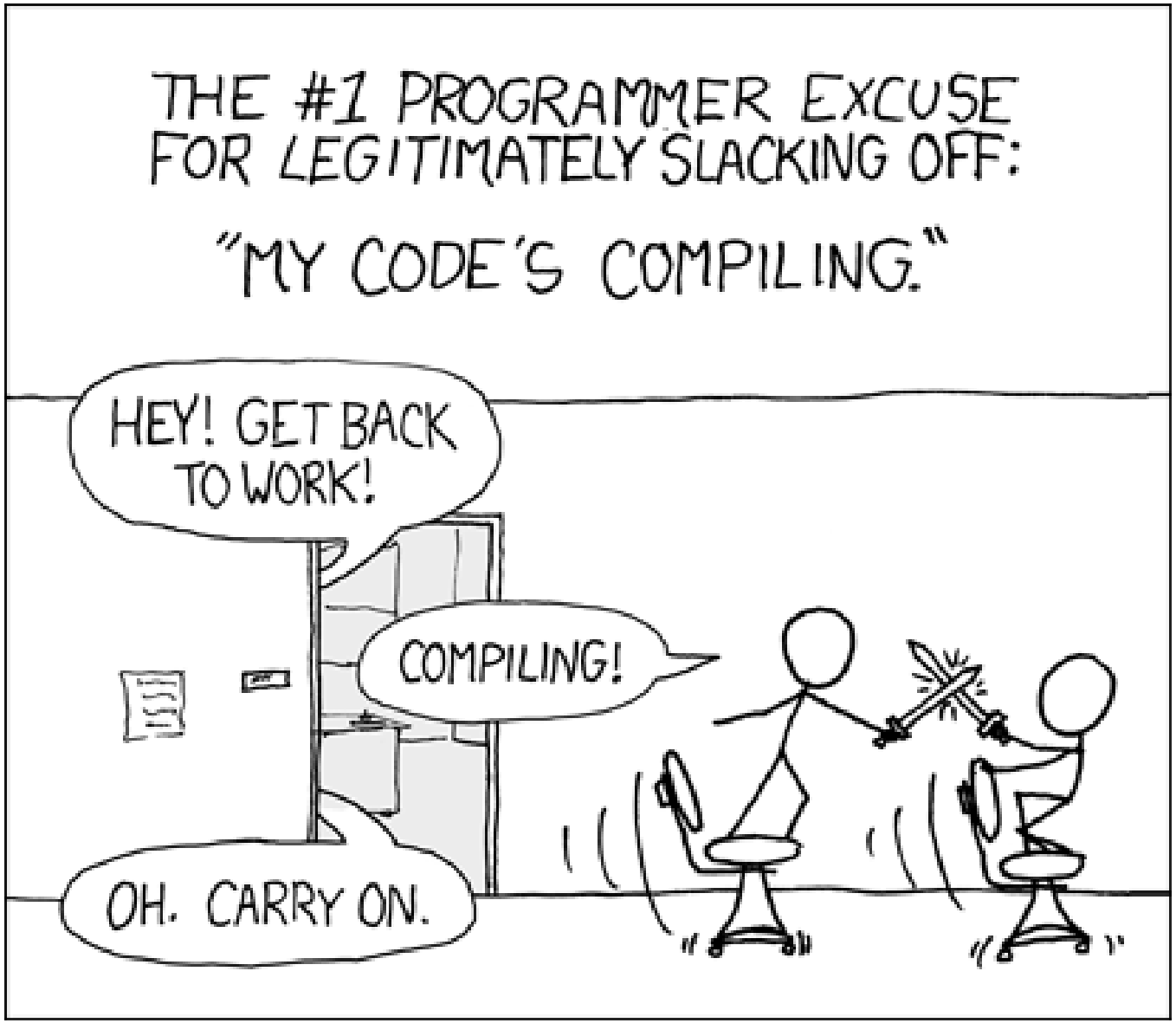


http://superoptimization.org/wiki/Superoptimizing_Compilers

Superoptimization

- requires detailed knowledge of the target architecture, including all side-effects

- takes a lot of time
  - unlike new patterns, this is actual, real compile time



https://xkcd.com/303/

Superoptimization

- requires detailed knowledge of the target architecture, including all side-effects
- takes a lot of time
  - unlike new patterns, this is actual, real compile time
- requires checking that generated sequences are semantically identical
- used only for really small performance sensitive kernels

And for finding new peephole patterns…

Further Reading

https://eli.thegreenplace.net/2012/11/24/life-of-an-instruction-in-llvm/ - don't be scared:)

https://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-171.html - The DENALI superoptimizer