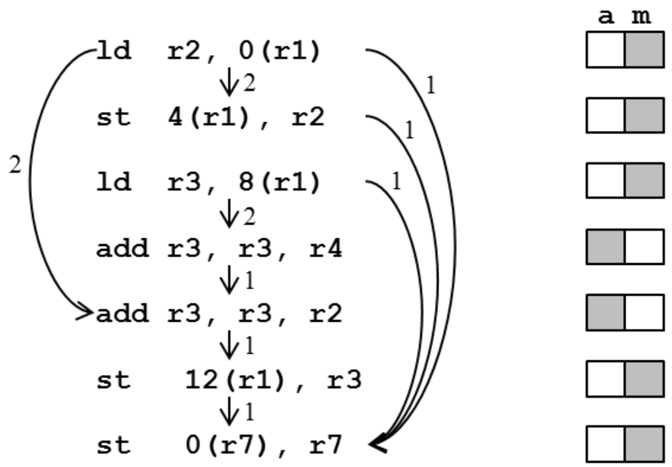
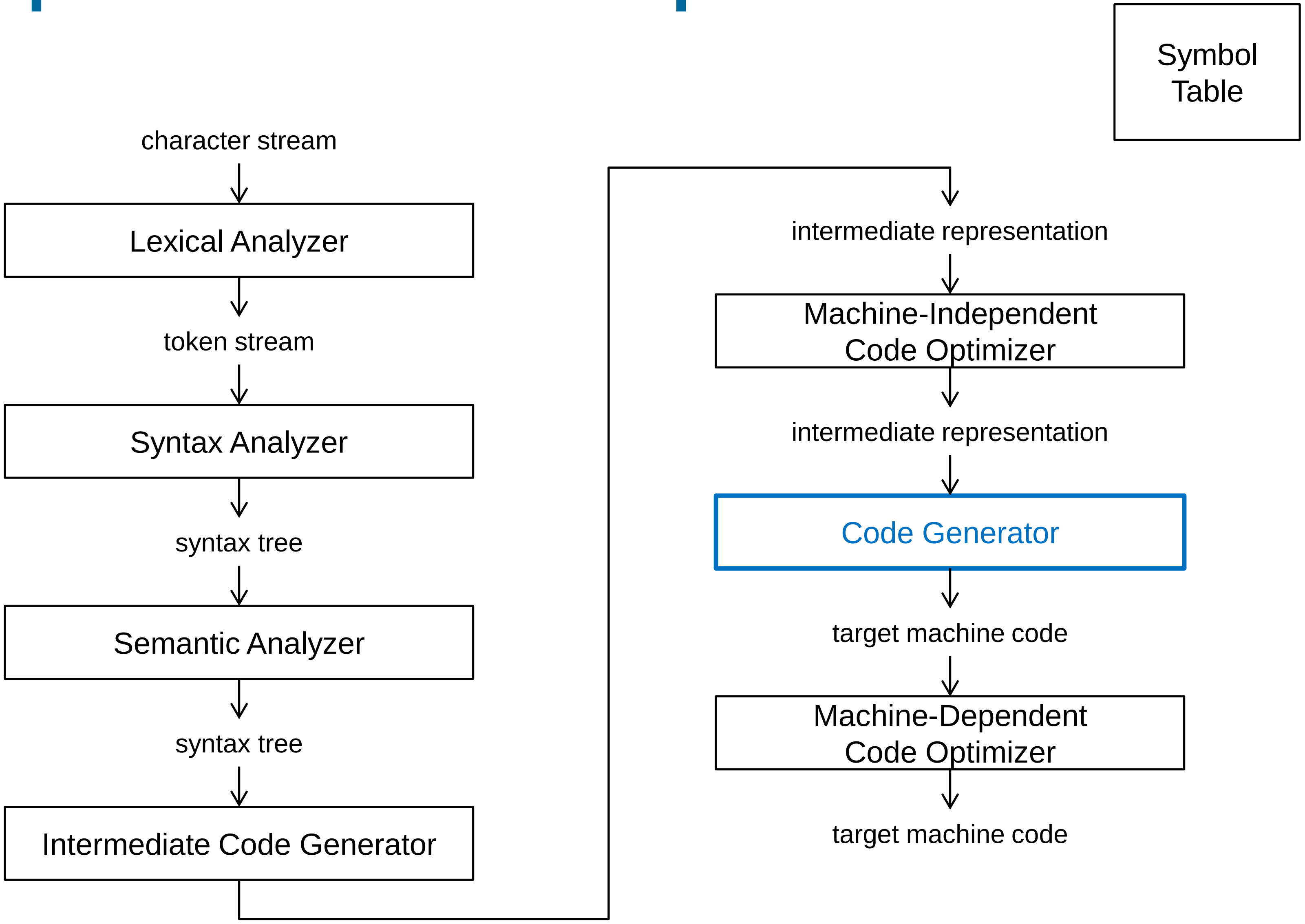
Code Generation Overview



Recap: Phases of a Compiler



Code Generation

- Code Generation is one of the final phases of a compiler
  - input: (machine-independent) IR
  - output: target machine code

- Code Generation typically consists of three phases
  - Instruction Selection
  - Register Allocation
  - Instruction Scheduling

- Different phase ordering can lead to better/worse code
  - computing the optimal machine code for a given source program is not feasible
  - we will use lots of heuristics
  - register allocation and instruction scheduling strongly depend on each other
    - both orderings (RA→IS, IS→RA) exist
    - as do techniques combining RA & IS

Code Generation

- Code Generator Input
  - we assume TAC, but the methods can also be applied to different forms of intermediate representations (AST, DAG, bytecodes, …)
  - all checks are done; i.e., the program is correct

- Code Generator Output
  - difficulty and methods heavily depend on the target machine
    - RISC
      - simple ISA, many registers, three-address instructions
    - CISC
      - complex ISA, few registers, two-address instructions
    - stack-based machines
      - push operands on stack; implicitly fetched by instructions

Code Generation

Instruction Selection

Instruction Selection Overview

- Instruction Selection
  determine a mapping from instructions in the IR to actual target machine instructions

- Instruction selection can range from rather simple to very complex
  - level of IR
    - high-level: translated using code templates
    - low-level: almost direct translation
  - target machine
    - uniformity and completeness
  - code quality
    - instructions with different latencies
    - instructions with different resource requirements

A Simple Instruction Selector

- Predefined code skeletons for each construct
  - example

$a = b + c \Longrightarrow$

```
movl b, %eax
addl c, %eax
movl %eax, a
```

  - often used in on-the-fly code generators
  - produces lots of redundant loads and stores:

$a = b + c$
$e = a + d \Longrightarrow$

```
movl b, %eax
addl c, %eax
movl %eax, a
movl a, %eax
addl d, %eax
movl %eax, e
```

(**movl %eax, a** and **movl a, %eax** in bold)

Instruction Selection

- In real processors, several valid instruction sequences compute the same result
  - selecting the one that
    - is the fasted
    - uses the least registers/energy
    - has the smallest code size
    - allows for more parallelization

    depends on the goal of code generation

  - examples: `add` vs `inc`, `div` vs `shift`

  - strategies:
    - code templates
    - tree-pattern matching
    - dynamic programming

Code Generation

Register Allocation

Register Allocation (RA)

- Registers are the fastest storage unit on the target machine
  - other storage units
    - Lx-cache: >>= 1 cycle
    - on-chip scratchpad memory: 1~10's of cycles
    - main memory: 10's – 100's of cycles
    - external storage: 10'000's of cycles

  - ISA may impose restrictions on register allocation
    - RISC machines: operands have to be in registers
    - hard-coded registers: IA32 `mull/divl`, ARM `bl`

Register Allocation

- Memory model of the compiler
  - memory-to-memory
    - all values reside in memory
    - register allocator
      - loads values into registers as needed
      - stores result back to memory
      - RA an optional optimization

  - register-to-register
    - all values reside in (virtual) registers
    - register allocator
      - evicts values to memory only if not enough physical registers are available
      - RA a necessity

Register Allocation Overview

- Register Allocation
  deciding which values are held in which registers

- Often performed by solving two subproblems
  1. register allocation
     which values to keep in registers at each program points
  2. register assignment
     map values that are to be kept in registers to specific (physical) registers

- Difficulties
  - NP-complete → resort to heuristics
  - spilling
  - register-usage conventions

Register Allocation

- Data structures required for RA
  - basic blocks
  - control flow
  - du-chains ("free" if IR in SSA form)

- Typically, register allocation is done on a global level
  - local would introduce lots and lots of moves between basic blocks
  - RA simply too important and needs to be done on a global level to get sufficient performance

Register Allocation Strategies

- Usage Count
  - observation:
    programs spend most of the time in loops → optimize register usage in loops
  - basic idea:
    - compute the benefit of keeping a variable in a register during execution of a loop
    - select the variables with the highest benefits

Register Allocation Strategies

- Usage Count
  - computation of benefits:
    - savings:
      - one unit for each use of v in a loop L that is not preceded by an assignment to v in the same block B
      - two units for each avoided store at the end of a block.
    - debit (neglected):
      - v live on entry to loop header: two units to load v into its register before entering the loop
      - v live on entry to some successor of any block in the loop: two units to store v before exiting the loop

$$benefit(v) = \sum_{blocks\ B\ in\ L}^{n} use(v,B) + 2 * live(x,B)$$

Register Allocation Strategies

- Usage Count
  - additional considerations for nested loops
    - start with innermost loops
    - compute benefits for outer loop given an assignment for the inner loop
    - often, loop count not known at compile time

Register Allocation Strategies

- Graph Coloring
  - deals with register allocation in the presence of spills

  - two passes
    1. perform instruction selection with an infinite number of (virtual) registers
    2. assign physical registers to virtual registers with
       - never use more than the available number of physical registers
       - minimize the number of spills

Register Allocation Strategies

- Graph Coloring
  - build interference graph
    G = (N, E)
    - nodes: virtual registers
    - edges: there exists an edge between two nodes N1 and N2 iff the life ranges of the associated virtual registers overlap (i.e., they are live at the same time)
  - assign a color (=physical register) to each node in such a way that no two nodes connected by an edge have the same color
    (“*k*-coloring” for *k* colors/registers)
  - graph coloring is NP-complete

Register Allocation Strategies

- Graph Coloring Heuristic
  - repeatedly eliminate nodes with less than k neighbors, leading to a reduced graph $G' = (N - \{n\}, E - \{\text{edges connected to } n\})$
    - a k-coloring for G' can easily be extended to G by assigning a color not used by any of the $<k$ neighbors of n.
  - if this process leads to the empty graph
    - in reverse order, assign a color to each node
    - done.
  - if some nodes are left, they will all have $>=k$ neighbors
    - spill one of the nodes, and continue the elimination step until no nodes are left.

  - Chaitin presented several heuristics on selecting spill nodes (see Register allocation and spilling via graph coloring, 1982, http://dl.acm.org/citation.cfm?id=806984)

Register Allocation Strategies

- Linear Scan
  - graph coloring has a runtime quadratic in the number of interference edges
  - not good for online RA

  - simple idea: linearly scan the interference graph from left to right
    - throw away expired intervals, add new intervals at every program point
    - as long as the number of live variables is <=k continue
    - if there is contention, spill the interval that ends the latest (i.e., furthest in the future)

  - complexity: O(N log k)

  - is fast, but introduces lots of spills
    - coalescing heuristics may help reducing the number of spills

Code Generation

Instruction Scheduling

Instruction Scheduling Overview

- Instruction Scheduling (or Ordering)
  pick an order for a series of instructions that gives the best performance

- Different orderings may require more/fewer registers

- Picking the best order is NP-complete

Instruction Scheduling

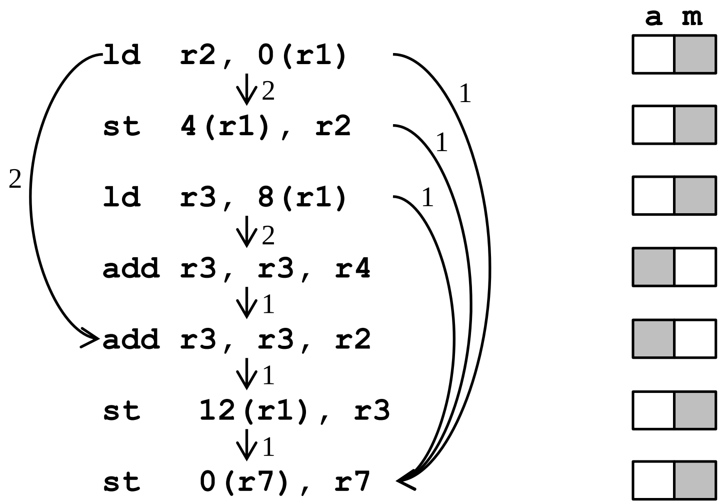
- A good order heavily depends on the target machine
  - available parallelism
    - super-scalar processors
    - VLIW, CGRA
  - processor pipeline
    - branch delays

- On a higher-level, the instruction schedule is constrained by
  - control dependences
  - data dependences

Instruction Scheduling

- Scheduling Instructions for a single basic block
  - NP-complete, but in practice, simple algorithms give good results

- Prevalent technique: List Scheduling
  - create data-dependence graph G = (N, E)
    N: instructions
    E: data-dependences, labeled with the instruction delay
  - create resource reservation table



```
ld   r2, 0(r1)
st   4(r1), r2
ld   r3, 8(r1)
add  r3, r3, r4
add  r3, r3, r2
st   12(r1), r3
st   0(r7), r7
```

Instruction Scheduling

- List Scheduling
  - visit each node in a prioritized topological order and issue the instruction
  - no backtracking

- Prioritized topological ordering?
  - no resource constraints: critical path first
  - no data dependences: schedule instructions with highest overuse first
  - lots of variants

- Further considerations
  - ASAP
  - ALAP

Instruction Scheduling

- Global Code Scheduling
  - consider more than one basic block at a time
    - i.e., a loop
- Techniques in global code scheduling
  - code motion
    - up, down: moving operations up, down between basic blocks
      - constraints imposed by dominator relation

Instruction Scheduling

- Global Code Scheduling Algorithms
  - region-based scheduling
    - based on a trace, a number of blocks are scheduled together, applying up/downwards code motion with compensation code
    - superblocks, hyperblocks
  - software pipelining
    - overlapping several iterations of one loop in software
    - issues here:
      - register allocation
      - resource usage
    - important subclass: modulo scheduling