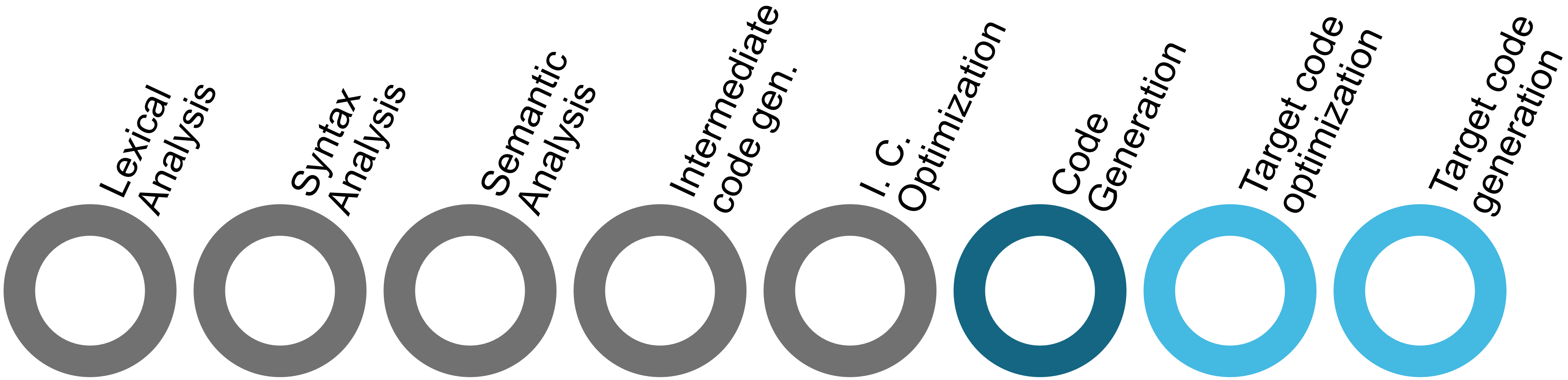
Compiler Design

Memory Management, Runtime Environments, Code Generation

Dr. Nicolai Stawinoga, Dr. Biagio Cosenza | TU Berlin | Wintersemester 2022-23

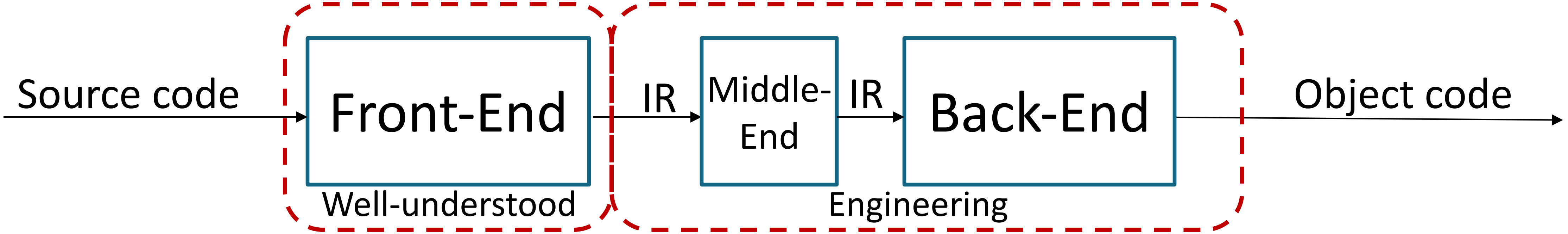
Where are we?



Lexical Analysis
Syntax Analysis
Semantic Analysis
Intermediate code gen.
I. C. Optimization
Code Generation
Target code optimization
Target code generation

Compiler Back-end

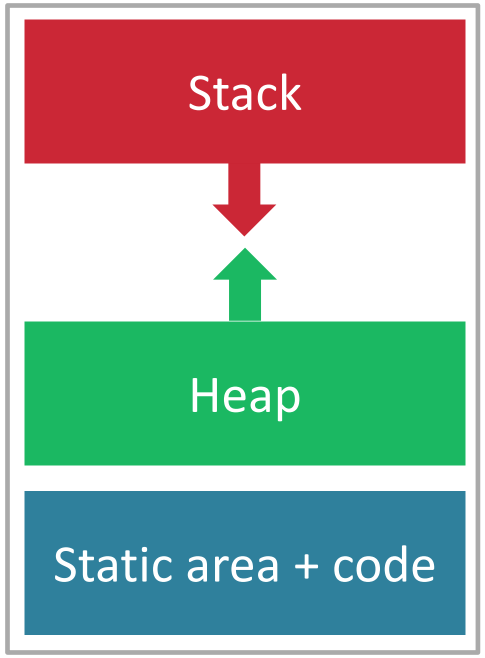
- We crossed the dividing line between the application of well-understood technology and fundamental issues of design and engineering
  - ➢the second half contains more open problems, more challenges, and more gray areas that the first half
- This is compilation as opposed to parsing or translation
  - ➢engineering as opposed to theory: imperfection, trade-off, constraints, optimization
  - ➢need to manage target machine resources



Memory Organization

- The memory used by a program can be allocated in different areas

1. Static area: for static and global variables
   - addresses are allocated statically (at compile-time)
2. Stack: for procedure local variables
   - in LIFO order
   - put them in the activation record if: sizes are fixed and values are not preserved
3. Heap: for dynamically allocated variables
   - usually allocated and deallocated explicitly
   - handled with pointers



The Procedure

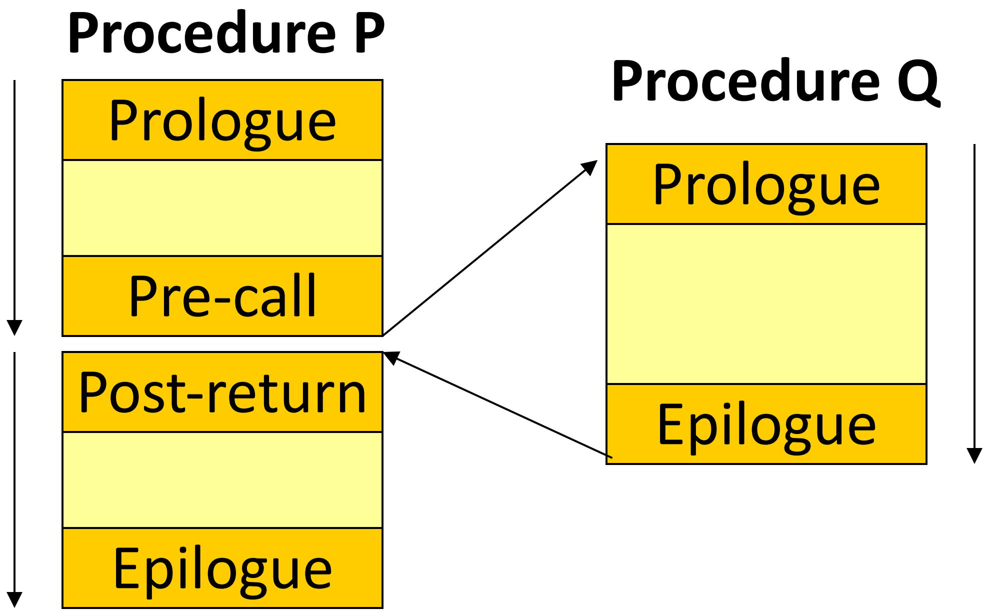
- Procedures are the key to building large systems; they provide
  - control abstraction: well-defined entries & exits
  - name space: has its own protected name space
  - external interface: access is by name & parameters
- Requires system-wide contract
  - broad agreement on memory layout, protection, etc…
  - must involve compiler, architecture, OS
- Establishes the need for private context
  - create a run-time “record” for each procedure to encapsulate information about control & data abstractions
- Separate compilation
  - allows us to build large systems; keeps compile-time reasonable

The Procedure: A More Abstract View

- A procedure is a collection of abstract concepts
- Underlying hardware supports little of this
  - well-defined entries and exits: mostly name-mangling
  - call/return mechanism: often done in software
  - name space, nested scopes: hardware does not understand them!
  - interfaces: need to be specified
- The procedure abstraction is an abstraction to allow the collaboration between OS & compiler

The Linkage Convention

- Procedures have well-defined control-flow behaviour:
  - a protocol for passing values and program control at procedure call and return is needed
  - the linkage convention ensures that procedures inherit a valid run-time environment and that they restore one for their parents
- Linkages execute at run-time
- Code to make the linkage is generated at compile-time



Technische Universität Berlin

Storage Organisation: Activation Records

- Local variables require storage during the lifetime of the procedure invocation at run-time
- The compiler arranges to set aside a region of memory for each individual call to a procedure (run-time support): activation record (also known as stack frame)

AR pointer →

parameters	
register save area	
return value	
return address	→ Address to resume caller
access link	→ Help with non-local access
caller’s AR	→ Pointer to caller’s activation record
local variables & temporaries	

In general, the compiler is free to choose any convention for the AR. The manufacturer may want to specify a standard for the architecture.

AES Embedded Systems Architecture

Procedure Linkages

- The procedure linkage convention is a machine-dependent contract between the compiler, the OS and the target machines to divide clearly responsibility

Caller (pre-call)
- allocate AR
- evaluate and store parameters
- store return address
- store self's AR pointer
- set AR pointer to child
- jump to child

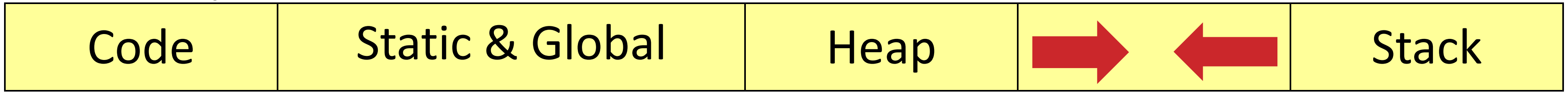
Callee (prologue)
- save registers, state
- extend AR for local data
- get static data area base address
- initialise local variables
- fall through to code

Callee (epilogue)
- store return value
- restore registers, state
- unextend basic frame
- restore parent's AR pointer
- jump to return address

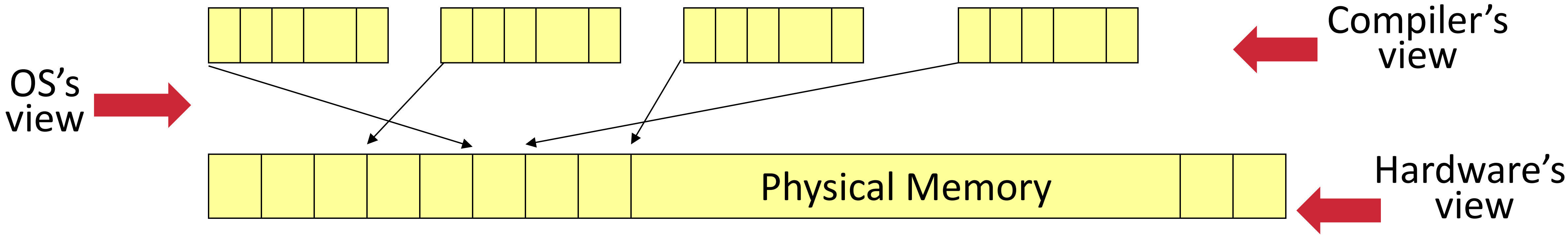
Caller (post-return)
- copy return value
- deallocate callee's AR
- restore parameters (if used for call-by reference)

Placing Runtime Run-time Data Structures

- Single logical address space



- Code, static, and global data have known size
- Heap & stack grow towards each other
  - ➢ From the compiler's perspective, the logical address space is the whole picture



Activation Record Details

- How does the compiler find the variables?
  - ➢they are offsets from the AR pointer
  - ➢variable length-data: if AR can be extended, put it below local variables; otherwise put on the heap
- Where do activation records live?
  - ➢if it makes no calls (leaf procedure - hence, only one can be active at a time), AR can be allocated statically
  - ➢place in the heap, if it needs to be active after exit (e.g., may return a pointer that refers to its execution state)
  - ➢otherwise place in the stack (this implies: lifetime of AR matches lifetime of invocation and code normally executes a “return”)
  - ➢(in decreasing order of efficiency: static, stack, heap)

Establishing Addressability

- Local variables of current procedure
  - ➢ if it is in the AR: use AR pointer and load as offset
  - ➢ if in the heap: store in the AR a pointer to the heap (double indirection)
  - ➢ (both the above need offset information)
  - ➢ if in a register: well, it is there!
- Global and static variables
  - ➢ use a relocatable (by the OS's loader) label (no need to emit code to determine address at run-time)
- Local variables of other procedures
  - ➢ need to retrieve information from the "other" procedure's AR

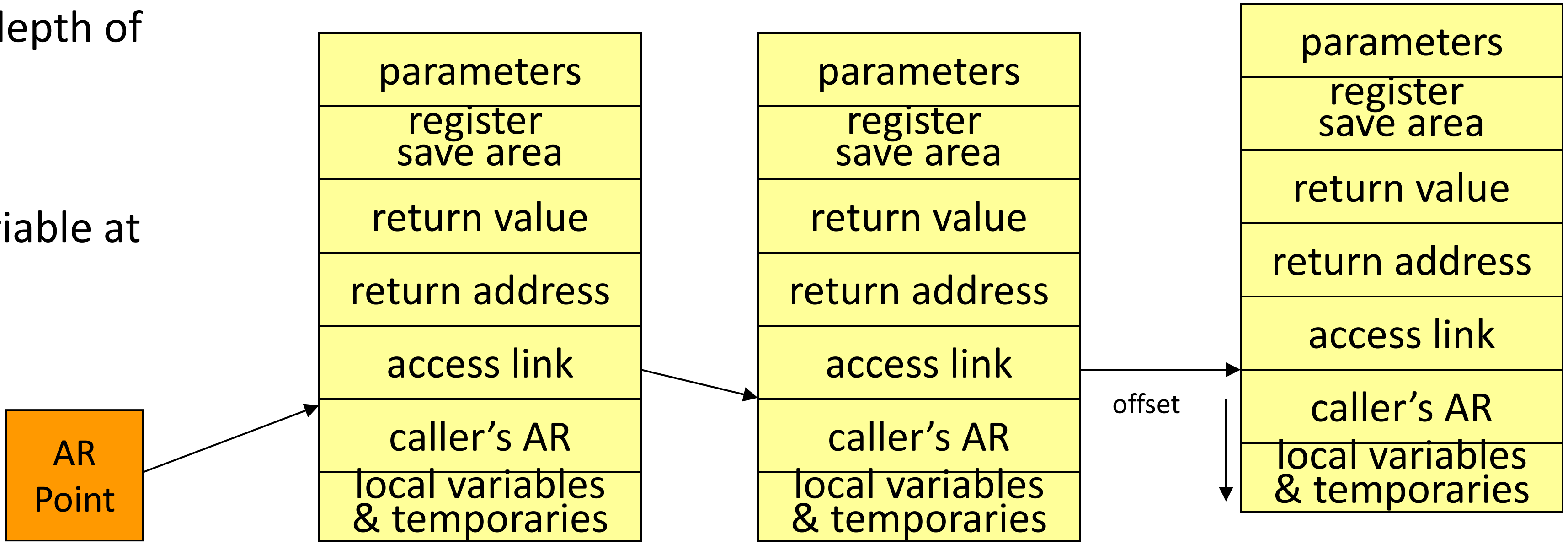
Addressing Non-local Data

- In a language that supports nested lexical scopes, the compiler must provide a mechanism to map variables onto addresses
- The compiler knows current level of lexical scope and of variable in question and offset (from the symbol table)
- Needs code to
  - track lexical ancestry (not necessarily the caller) among ARs
  - interpret difference between levels of lexical scope and offset
- Two basic mechanisms
  - access links
  - global display

```
let function f():int = let
  var a:=5
  function g(y:int):int = let
    var b:=10
    function h(z:int):int =
      if z > 10 then h(z / 2)
      else z + b * a
    in
      y + a + h(16)
      end
  in
      g(10)
    end
in f() end
```

Access Links

- Idea: Each AR contains a pointer to its lexical ancestor
  - compiler needs to emit code to find lexical ancestor (if caller's scope=callee's scope+1 then it is the caller; else walk through the caller's ancestors)
  - cost of access depends on depth of lexical nesting
- Example:
  - (current level=2): needs variable at level=0, offset=16:
  - `load r1,(ARP-4);`
  - `load r1,(r1-4);`
  - `load r2,(r1+16)`

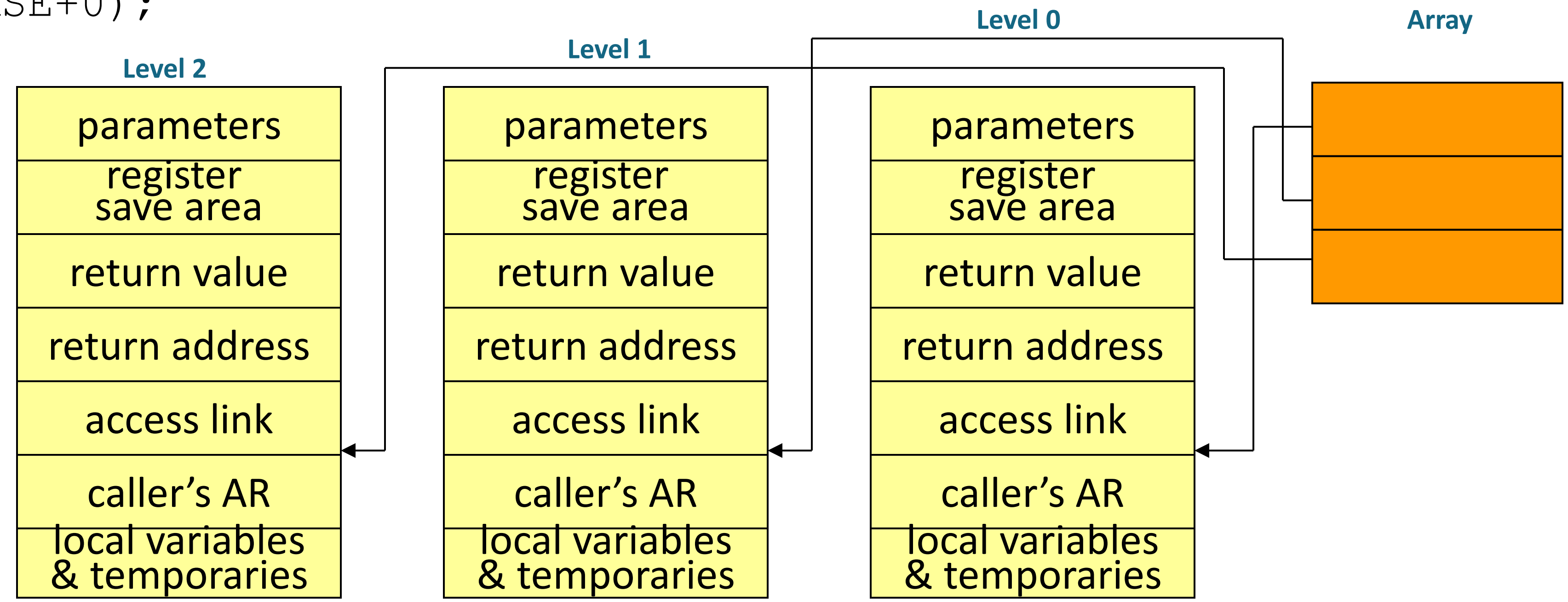


Global Display

- Idea: keep a global array to hold ARPs for each level
  - ➢compiler needs to emit code (when calling and returning from a procedure) to maintain the array
  - ➢cost of access is fixed (table lookup + AR)
- Example:
  - ➢(current level=2): needs variable at level=0, offset=16:

```
load r1,(DISPLAY_BASE+0);
load r2,(r1+16)
```

- Display vs access links trade-off
  - ➢conventional wisdom: use access links when tight on registers; display when lots of registers



Inlining (and Outlining)

- The compiler needs to emit code
  - for each call to a procedure to take into account (at run-time) procedure linkage
  - to provide (at run-time) addressability for variables of other procedures
- Inlining: the compiler can avoid some of the problems related to procedures by substituting a procedure call with the actual code for the procedure
  - there are advantages from doing this, but it may not be always possible (can you see why?) and there are disadvantages too
  - typical approach use SCC (strongly connected component) between functions
- Outlining: reverse of inlining: replace a block of consecutive statements with a function call to a new function containing those statements
  - Useful to reduce code size

Dynamic Memory Allocation (Heap)

- Managing the heap is difficult
  - ➢ irregular lifetime of the block allocated
- Memory deallocation, two approaches
  - ➢ explicit: when the programming language offers a way to deallocate memory blocks as being free
    - E.g., `delete` (C++) and `free` (C)
    - Common problems
      - memory can be freed too early, leading to dangling pointers, data corruption, ect..
      - memory can be freed too late or never, which leads to memory leaks
      - memory can be freed twice: double free
  - ➢ implicit: a runtime system infers what data requires deallocation, e.g., by finding which allocated blocks are not reachable anymore
    - Garbage Collection (GC): set of techniques which automatically reclaim objects which are not reachable anymore

Garbage Collection (GC)

- Automatic memory management
  - ➢ the GC reclaims memory occupied by objects that are no longer in use
  - ➢ such objects are called garbage
- Two steps
  1. Scan objects in memory, identify objects that cannot be accessed (now, or in the future)
  2. Reclaim these garbage objects
- Approaches to GC
  - ➢ Reference counting
  - ➢ Mark & Sweep
  - ➢ Copying Collection
  - ➢ Generational Collection

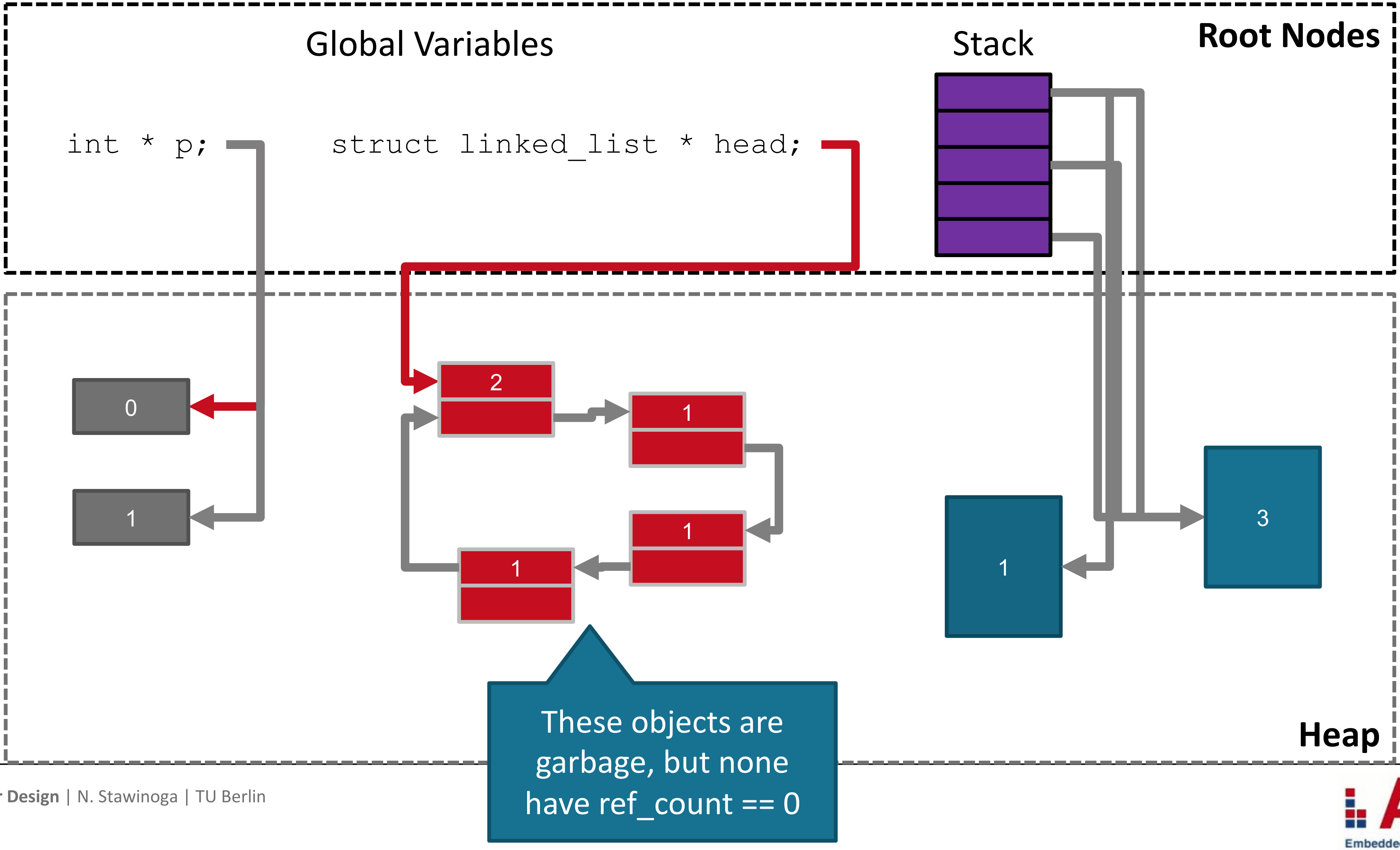
Garbage Collection with Reference Counting

- Reference counting
  - ➢ track the number of outstanding pointers referring to an object
    - Each object includes a ref_count, If ref_count is 0, it is garbage
  - ➢ garbage collection: when there is no space, stop execution and discover objects that can be reached from pointers stored in program variables; unreachable space is recycled
  - ➢ problem with cyclic structures
    - Not guaranteed to free all garbage objects

Garbage Collection with Mark & Sweep

- Key idea: periodically scan all objects for reachability
  - start at the roots
  - traverse all reachable objects, mark them
  - all unmarked objects are garbage

Reference Counting Example

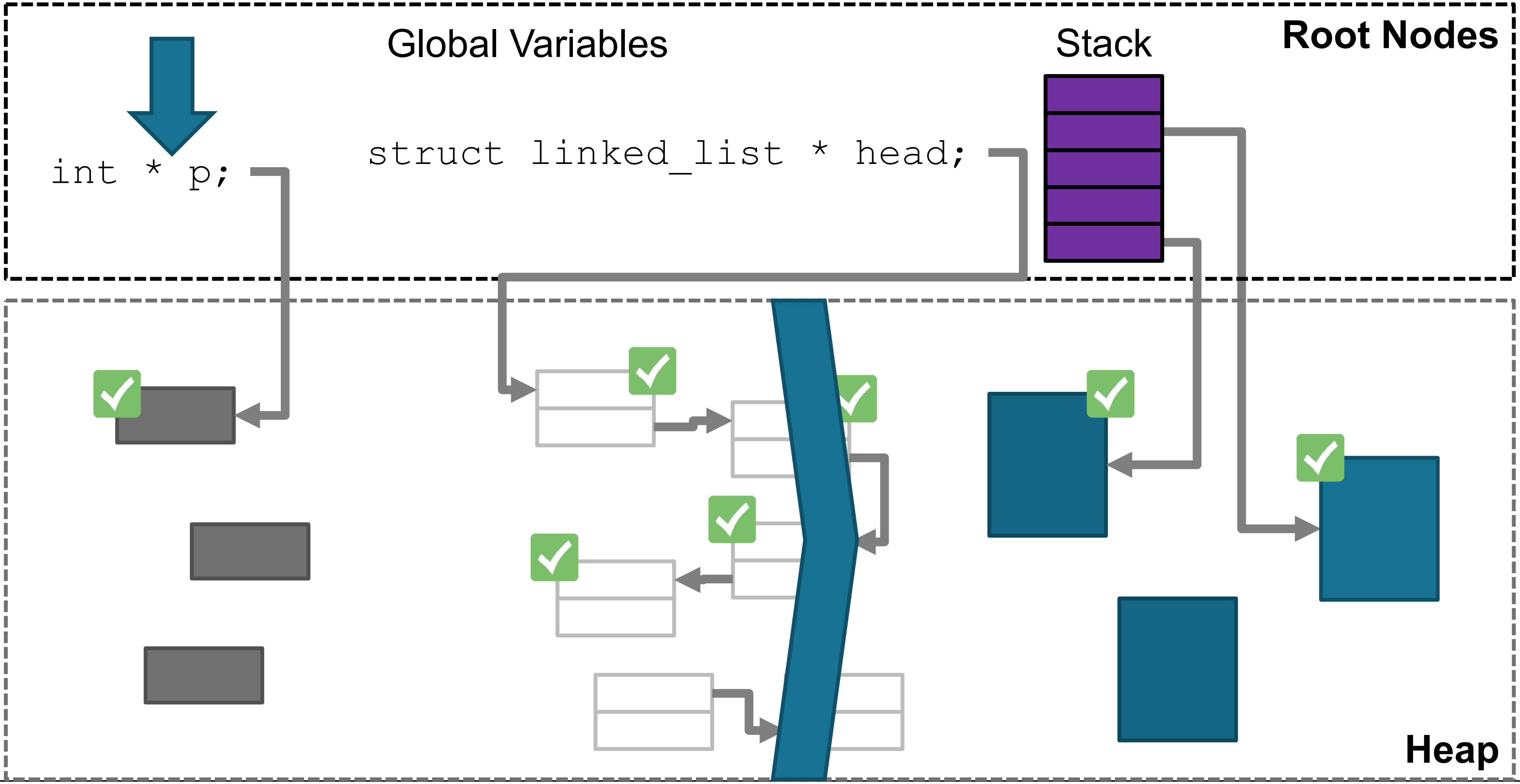


Note for printed slides: this and the following slides have animations.

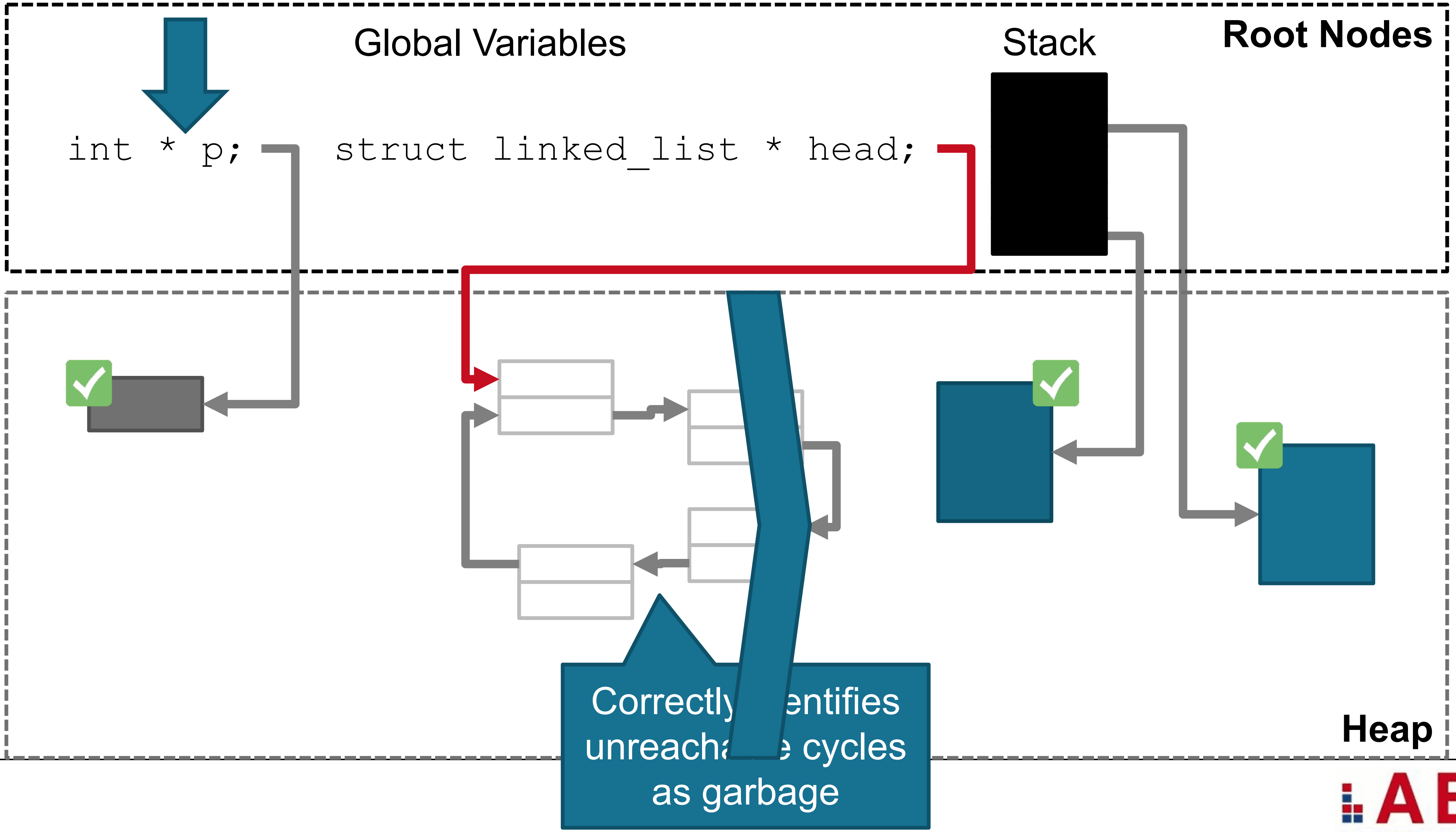
Red lines are first created, and later removed.

In this case, the four red nodes become unreachable from head after the removal.

Mark & Sweep Example



Mark & Sweep: an Example with Cycles



Mark & Sweep: Summary

- Good
  - ➢ Overcomes the weakness of reference counting
  - ➢ Fairly easy to implement and conceptualize
  - ➢ Guaranteed to free all garbage objects
- Bad
  - ➢ Mark & sweep is CPU intensive
    - traverses all objects reachable from the root
    - scans all objects in memory freeing unmarked objects
  - ➢ Naïve implementations may “pause the system” before collecting
    - threads cannot run in parallel with the GC
    - all threads get stopped while the GC runs

Other approaches: Copy Collection, Generational Collection

Explicit memory allocation vs GC Summary

Explicit

- Advantages
  - typically faster than GC
  - no GC “pauses” in execution
  - more efficient use of memory
- Disadvantages
  - more complex for programmers
  - tricky memory bugs
  - dangling pointers
  - double-free
  - memory leaks
  - bugs may lead to security vulnerabilities

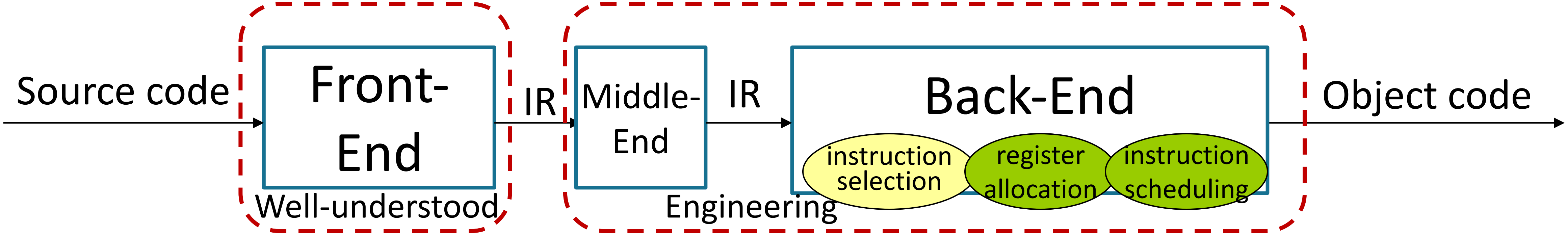
Garbage Collection

- Advantages
  - much easier for programmers
- Disadvantages
  - typically slower than explicit alloc/dealloc
  - good performance requires careful tuning of the GC
  - less efficient use of memory
  - complex runtimes may have security vulnerabilities
    - JVM gets exploited all the time

Summary

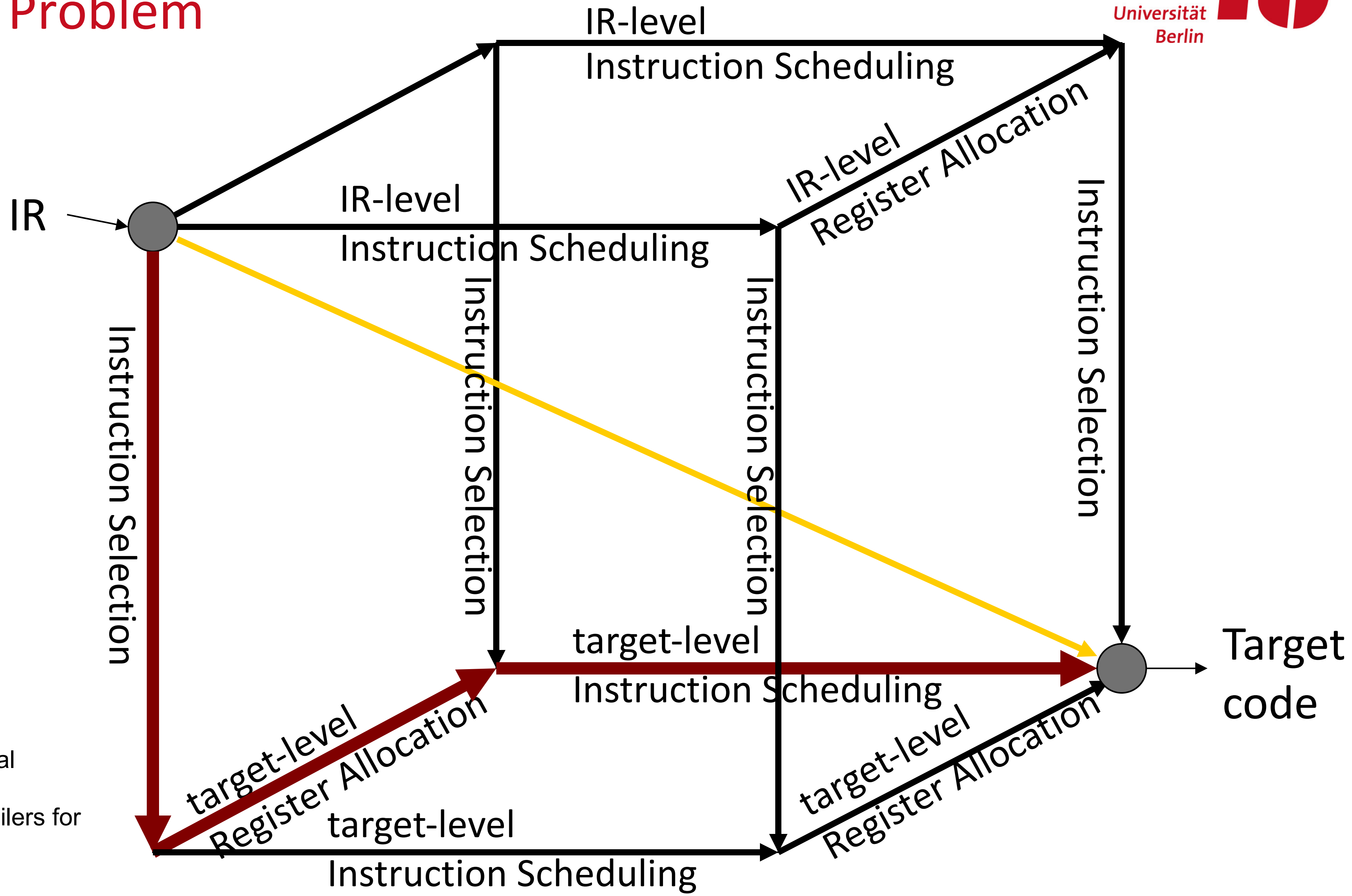
- Procedure, linkage convention, activation record
- Access links vs global display
- Inlining, outlining
- Dynamic memory allocation, garbage collection, reference counting, mark and sweep
- Readings
  - ALSU Section 7.1, 7.2, 7.3

Code Generation & Instruction Selection



- We assume the following model for code generation
  - instruction selection: mapping IR into assembly code
  - register allocation: decide which values will reside in registers
  - instruction scheduling: reorder operations to hide latencies
- Conventional wisdom says that we lose little by solving these (NP-complete) problems independently

Code Generation as a Phase-decoupled Problem



Source: C.Kessler, A.Bednarski; Optimal Integrated Code Generation for VLIW architectures; 10th Workshop on Compilers for Parallel Computers, 2003

Instruction Selection

- Characteristics
  - use some form of pattern matching
  - can make locally optimal choices, but global optimality is NP-complete
  - assume enough registers.
- Assume a RISC-like target language
- Recall
  - modern processors can issue multiple instructions at the same time
  - instructions may have different latencies: e.g., add: 1 cycle; load 1-3 cycles; imult: 3-16 cycles, etc
- Instruction Latency: length time elapsed from the time the instruction was issued until the time the results can be used

Example: A Very Basic Instruction Set

```
load r1, @a     ; load register r1 with the memory value for a
load r1, [r2]   ; load register r1 with the memory value pointed by r2
mov r1, 3       ; r1=3
add r1, r2, r3  ; r1=r2+r3
```

- (same for mult, sub, div)

```
shr r1, r1, 1   ; r1=r1>>1 (shift right; r1/2)
store r1        ; store r1 in memory
nop             ; no operation
```

Code Generation for Arithmetic Expressions

- Adopt a simple tree-walk scheme; emit code in post-order

```
expr(node)
{ int result, t1, t2;
  switch(type(node))
  { case *,/,+,-:
      t1=expr(left child(node));
      t2=expr(right child(node));
      result = NextRegister();
      emit(op(node),result,t1,t2);
      break;
    case IDENTIFIER:
      t1=base(node); t2=offset(node);
      result = NextRegister();
      emit(…) /* load IDENTIFIER */
      break;
    case NUM:
      result = NextRegister();
      emit(load result, val(node));
      break;
  }
  return result;
}
```

Example: `x+y:`

```
load r1, @x
load r2, @y
add r3,r1,r2

(load r1,@x would
involve a load from
address base+offset)
```

Issues with arithmetic expressions (1)

- What about values already in registers?
  - ➢ modify the IDENTIFIER case
- Why the left subtree first and not the right?
  - ➢ (cf. `2*y+x; x-2*y; x+(5+y)*7`): the most demanding (in registers) subtree should be evaluated first
- 2nd pass to minimize register usage/improve performance
- The compiler can take advantage of commutativity and associativity to improve code (but not for floating-point operations)

Issues with arithmetic expressions (2)

- Observation: on most processors, the cost of a `mult` instruction might be several cycles; the cost of shift and add instructions is, typically, 1 cycle
- Problem: generate code that multiplies an integer with an unknown using only shifts and adds
- E.g.:
  - `325*x = 256*x+64*x+4*x+x  or  (4*x+x)*(64+1)` (Sparc)
- For division, say x/3, we could compute 1/3 and perform a multiplication (using shifts and adds)… but this is getting complex!

Array References

- Agree to a storage scheme
  - Row-major order: layout as a sequence of consecutive rows: `A[1,1], A[1,2], A[1,3], A[2,1], A[2,2], A[2,3]`. Example: C/C++/Objective C, Mathematica, Pascal, C#
  - Column-major order: layout as a sequence of consecutive columns: `A[1,1], A[2,1], A[1,2], A[2,2], A[3,1], A[3,2]`. Example: Fortran, OpenGL, Matlab, R, GNU Octave, Julia
  - Indirection vectors: vector of pointers to pointers to … values; best storage is application dependent; not amenable to analysis. Example: **Iliffe vectors** used in Java, Scala and Swift
- Referencing an array element, where `w=sizeof(element)`
  - row-major, 2d: $\mathtt{base+((i_1-low_1)*(high_2-low_2+1)+i_2-low_2)*w}$
  - column-major, 2d: $\mathtt{base+((i_2-low_2)*(high_1-low_1+1)+i_1-low_1)*w}$
  - general case for row-major order: $\mathtt{((\ldots(i_1 n_2+i_2)n_3+i_3)\ldots)n_k+i_k)*w+base-w*((\ldots((low_1*n_2)+low_2)n_3+low_3)\ldots)n_k+low_k)}$, where $\mathtt{n_i=high_i-low_i+1}$

Boolean and Relational Values

```
expr → not or-term | or-term
or-term → or-term or and-term | and-term
and-term → and-term and boolean | boolean
boolean → true | false | rel-term
rel-term → rel-term rel-op expr | expr
rel-op → < | > | == | != | >= | <=
```

- Evaluate using tree-walk-style generation. Two approaches for translation: numerical representation (0/1) or positional encoding
  - B or C and not D: r1=not D; r2=r1 and C; r3=r2 or B.
  - if (a<b) then … else… : comp rx,ra,rb; br rx L1, L2; L1: …code for then...; br L3; L2: …code for else...; L3: ...
- Short-circuit evaluation: the C expression (x!=0 && y/x>1) relies on short-circuit evaluation for safety

Control-Flow Statements

- If expr then stmt1 else stmt2
  1. Evaluate the expr
  2. If true, fall through to then; branch around else
  3. If false, branch to else; fall through to next statement

```
if not(expr) br L1; stmt1; br L2; L1: stmt2; L2: …
```

- While loop; for loop; or do loop
  1. Evaluate expr
  2. If false, branch beyond end of loop; if true, fall through
  3. At end, re-evaluate expr
  4. If true, branch to top of loop body; if false, fall through

```
if not(expr) br L2; L1: loop-body; if (expr) br L1; L2: …
```

- Case statement: evaluate; branch to case; execute; branch

Conclusion

- (Initial) code generation is a pattern matching problem
- Code generation involves three problems
  - Instruction selection
  - Register Allocation
  - Instruction scheduling

Trading register usage with performance

- Example: `w=w*2*x*y*z`

```
1.  load r1, @w
2.  mov  r2, 2
6.  mult r1,r1,r2
7.  load r2, @x
12.  mult r1,r1,r2
13.  load r2, @y
18.  mult r1,r1,r2
19.  load r2, @z
24.  mult r1,r1,r2
26.  store r1
```

```
1.  load r1, @w        ; load: 5 cycles
2.  load r2, @x
3.  load r3, @y
4.  load r4, @z
5.  mov  r5, 2         ; mov: 1 cycle
6.  mult r1,r1,r5      ; mult: 2 cycles
8.  mult r1,r1,r2
10.  mult r1,r1,r3
12.  mult r1,r1,r4
14.  store r1           ; store: 5 cycles
```

- Instruction scheduling (the problem): given a code fragment and the latencies for each operation, reorder the operations to minimize execution time (produce correct code; avoid spilling registers)

Summary

- Instruction Selection
- Code generation with arithmetic expressions, array, boolean, control-flow statements

- Readings
  - ALSU Chapter 8
  - Note: ALSU treats intermediate and machine code generation as two separate problems but follows a low-level IR; things may vary in reality