

Lecture 7

CIS 341: COMPILERS

Announcements

- HW2: X86lite
 - Available on the course web pages.
 - Due: TOMORROW at 11:59:59pm
 - NOTE: submission server was broken last night/this morning
 - It should now support Ocaml version 4.06.0
 - Only one group member needs to submit
 - Three submissions per group

Intermediate Representations

- IR1: Expressions
 - simple arithmetic expressions, immutable global variables
- IR2: Commands
 - global *mutable* variables
 - commands for update and sequencing
- IR3: Local control flow
 - conditional commands & while loops
 - basic blocks
- IR4: Procedures (top-level functions)
 - local state
 - call stack

Basic Blocks

- A sequence of instructions that is always executed starting at the first instruction and always exits at the last instruction.
 - Starts with a label that names the *entry point* of the basic block.
 - Ends with a control-flow instruction (e.g. branch or return) the “link”
 - Contains no other control-flow instructions
 - Contains no interior label used as a jump target
- Basic blocks can be arranged into a *control-flow graph*
 - Nodes are basic blocks
 - There is a directed edge from node A to node B if the control flow instruction at the end of basic block A might jump to the label of basic block B.

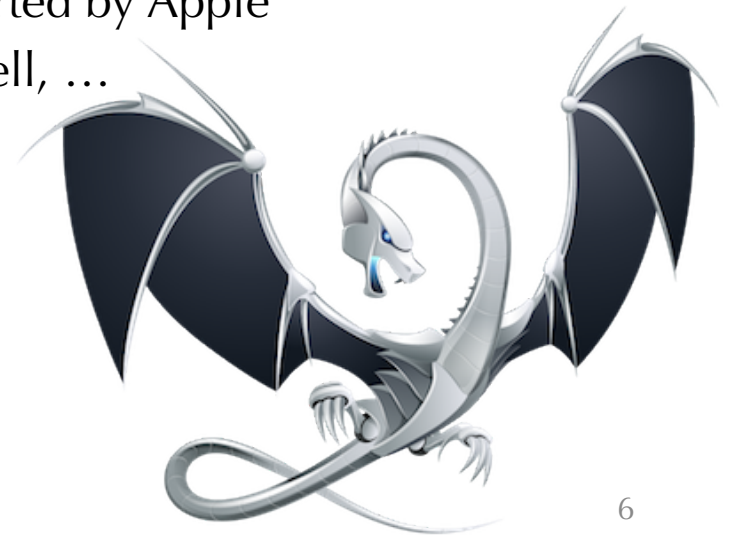


See llvm.org

LLVM

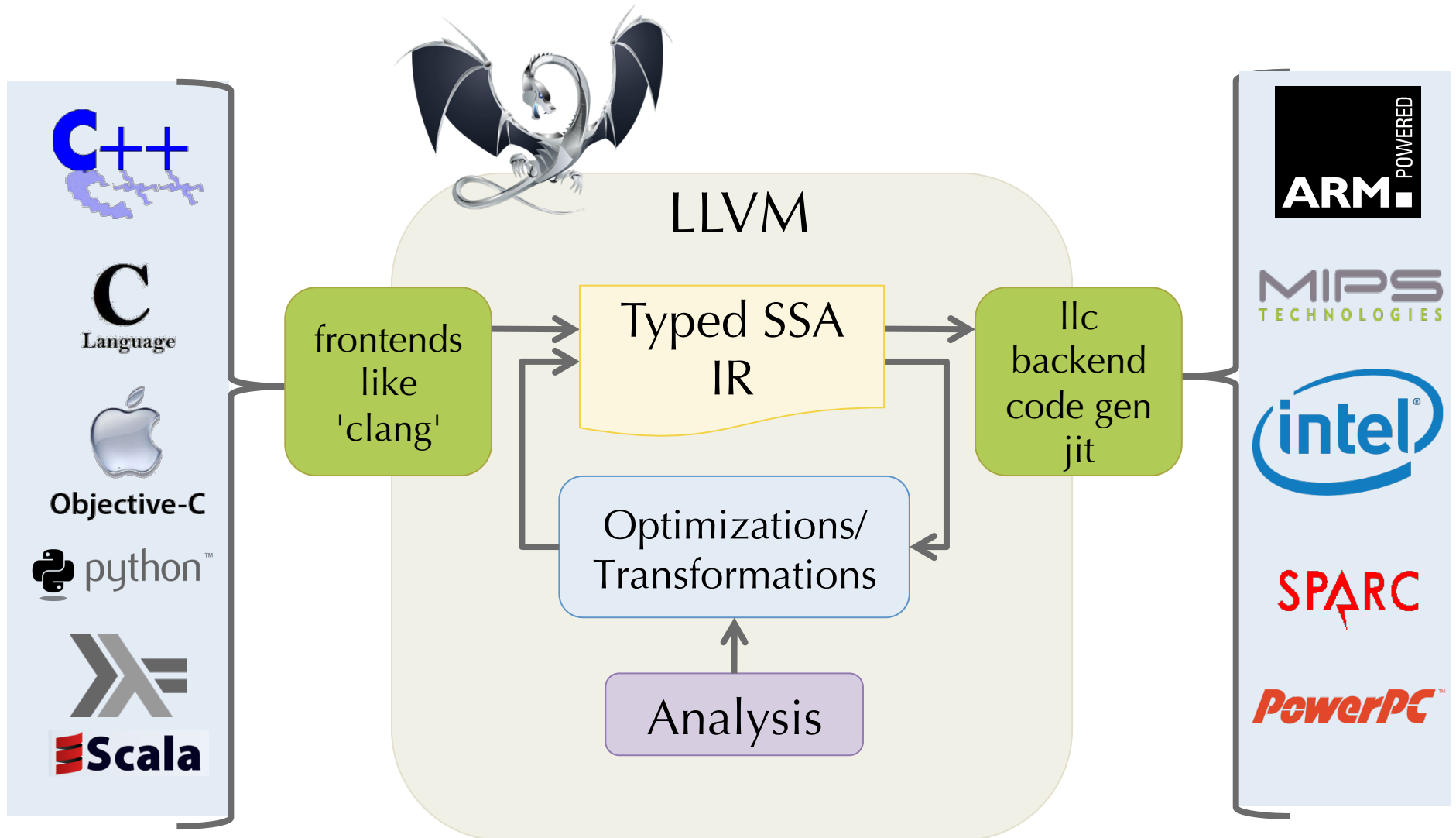
Low-Level Virtual Machine (LLVM)

- Open-Source Compiler Infrastructure
 - see llvm.org for full documentation
- Created by Chris Lattner (advised by Vikram Adve) at UIUC
 - LLVM: An infrastructure for Mult-stage Optimization, 2002
 - LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation, 2004
- 2005: Adopted by Apple for XCode 3.1
- Front ends:
 - llvm-gcc (drop-in replacement for gcc)
 - Clang: C, objective C, C++ compiler supported by Apple
 - various languages: Swift, ADA, Scala, Haskell, ...
- Back ends:
 - x86 / Arm / Power / etc.
- Used in many academic/research projects
 - Here at Penn: SoftBound, Vellvm



LLVM Compiler Infrastructure

[Lattner et al.]



Example LLVM Code

- LLVM offers a textual representation of its IR
 - files ending in .ll

factorial64.c

```
#include <stdio.h>
#include <stdint.h>

int64_t factorial(int64_t n) {
    int64_t acc = 1;
    while (n > 0) {
        acc = acc * n;
        n = n - 1;
    }
    return acc;
}
```



factorial-pretty.ll

```
define @factorial(%n) {
    %1 = alloca
    %acc = alloca
    store %n, %1
    store 1, %acc
    br label %start

start:
    %3 = load %1
    %4 = icmp sgt %3, 0
    br %4, label %then, label %else

then:
    %6 = load %acc
    %7 = load %1
    %8 = mul %6, %7
    store %8, %acc
    %9 = load %1
    %10 = sub %9, 1
    store %10, %1
    br label %start

else:
    %12 = load %acc
    ret %12
}
```


Real LLVM

- Decorates values with type information

i64

i64*

i1

- Permits numeric identifiers
- Has alignment annotations
- Keeps track of entry edges for each block:
preds = %5, %0

factorial.ll

```
; Function Attrs: nounwind ssp
define i64 @factorial(i64 %n) #0 {
    %1 = alloca i64, align 8
    %acc = alloca i64, align 8
    store i64 %n, i64* %1, align 8
    store i64 1, i64* %acc, align 8
    br label %2

; <label>:2                                ; preds = %5, %0
    %3 = load i64* %1, align 8
    %4 = icmp sgt i64 %3, 0
    br i1 %4, label %5, label %11

; <label>:5                                ; preds = %2
    %6 = load i64* %acc, align 8
    %7 = load i64* %1, align 8
    %8 = mul nsw i64 %6, %7
    store i64 %8, i64* %acc, align 8
    %9 = load i64* %1, align 8
    %10 = sub nsw i64 %9, 1
    store i64 %10, i64* %1, align 8
    br label %2

; <label>:11                                ; preds = %2
    %12 = load i64* %acc, align 8
    ret i64 %12
}
```

Example Control-flow Graph

```
define @factorial(%n) {
```

entry:

```
%1 = alloca  
%acc = alloca  
store %n, %1  
store 1, %acc  
br label %start
```

loop:

```
%3 = load %1  
%4 = icmp sgt %3, 0  
br %4, label %then, label %else
```

body:

```
%6 = load %acc  
%7 = load %1  
%8 = mul %6, %7  
store %8, %acc  
%9 = load %1  
%10 = sub %9, 1  
store %10, %1  
br label %start
```

post:

```
%12 = load %acc  
ret %12
```

```
}
```

LL Basic Blocks and Control-Flow Graphs

- LLVM enforces (some of) the basic block invariants syntactically.
- Representation in OCaml:

```
type block = {  
    insns : (uid * insn) list;  
    term  : (uid * terminator)  
}
```

- A *control flow graph* is represented as a list of labeled basic blocks with these invariants:
 - No two blocks have the same label
 - All terminators mention only labels that are defined among the set of basic blocks
 - There is a distinguished, unlabeled, entry block:

```
type cfg = block * (lbl * block) list
```

LL Storage Model: Locals

- Several kinds of storage:
 - Local variables (or temporaries): `%uid`
 - Global declarations (e.g. for string constants): `@gid`
 - Abstract locations: references to (stack-allocated) storage created by the `alloca` instruction
 - Heap-allocated structures created by external calls (e.g. to `malloc`)
- Local variables:
 - Defined by the instructions of the form `%uid = ...`
 - Must satisfy the *single static assignment* invariant
 - Each `%uid` appears on the left-hand side of an assignment only once in the entire control flow graph.
 - The value of a `%uid` remains unchanged throughout its lifetime
 - Analogous to “`let %uid = e in ...`” in OCaml
- Intended to be an abstract version of machine registers.
- We’ll see later how to extend SSA to allow richer use of local variables
 - *phi nodes*

LL Storage Model: `alloca`

- The `alloca` instruction allocates stack space and returns a reference to it.
 - The returned reference is stored in local:
`%ptr = alloca typ`
 - The amount of space allocated is determined by the type
- The contents of the slot are accessed via the `load` and `store` instructions:

```
%acc = alloca i64           ; allocate a storage slot
store i64 341, i64* %acc    ; store the integer value 341
%x = load i64, i64* %acc    ; load the value 341 into %x
```

- Gives an abstract version of stack slots



STRUCTURED DATA

Compiling Structured Data

- Consider C-style structures like those below.
- How do we represent `Point` and `Rect` values?

```
struct Point { int x; int y; };

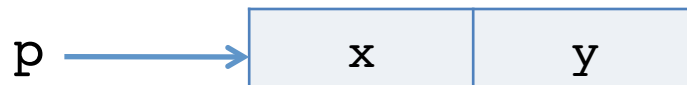
struct Rect  { struct Point ll, lr, ul, ur };

struct Rect mk_square(struct Point ll, int len) {
    struct Rect square;
    square.ll = square.lr = square.ul = square.ur = ll;
    square.lr.x += len;
    square.ul.y += len;
    square.ur.x += len;
    square.ur.y += len;
    return square;
}
```

Representing Structs

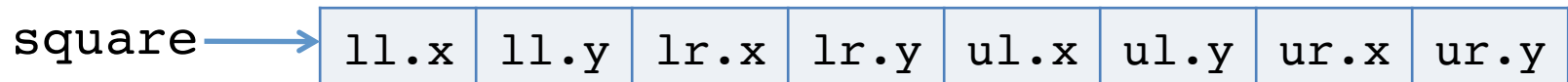
```
struct Point { int x; int y;};
```

- Store the data using two contiguous words of memory.
- Represent a `Point` value `p` as the address of the first word.



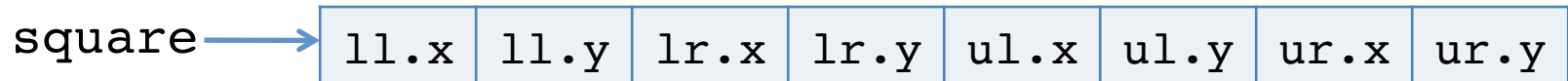
```
struct Rect { struct Point ll, lr, ul, ur };
```

- Store the data using 8 contiguous words of memory.



- Compiler needs to know the *size* of the struct at compile time to allocate the needed storage space.
- Compiler needs to know the *shape* of the struct at compile time to index into the structure.

Assembly-level Member Access

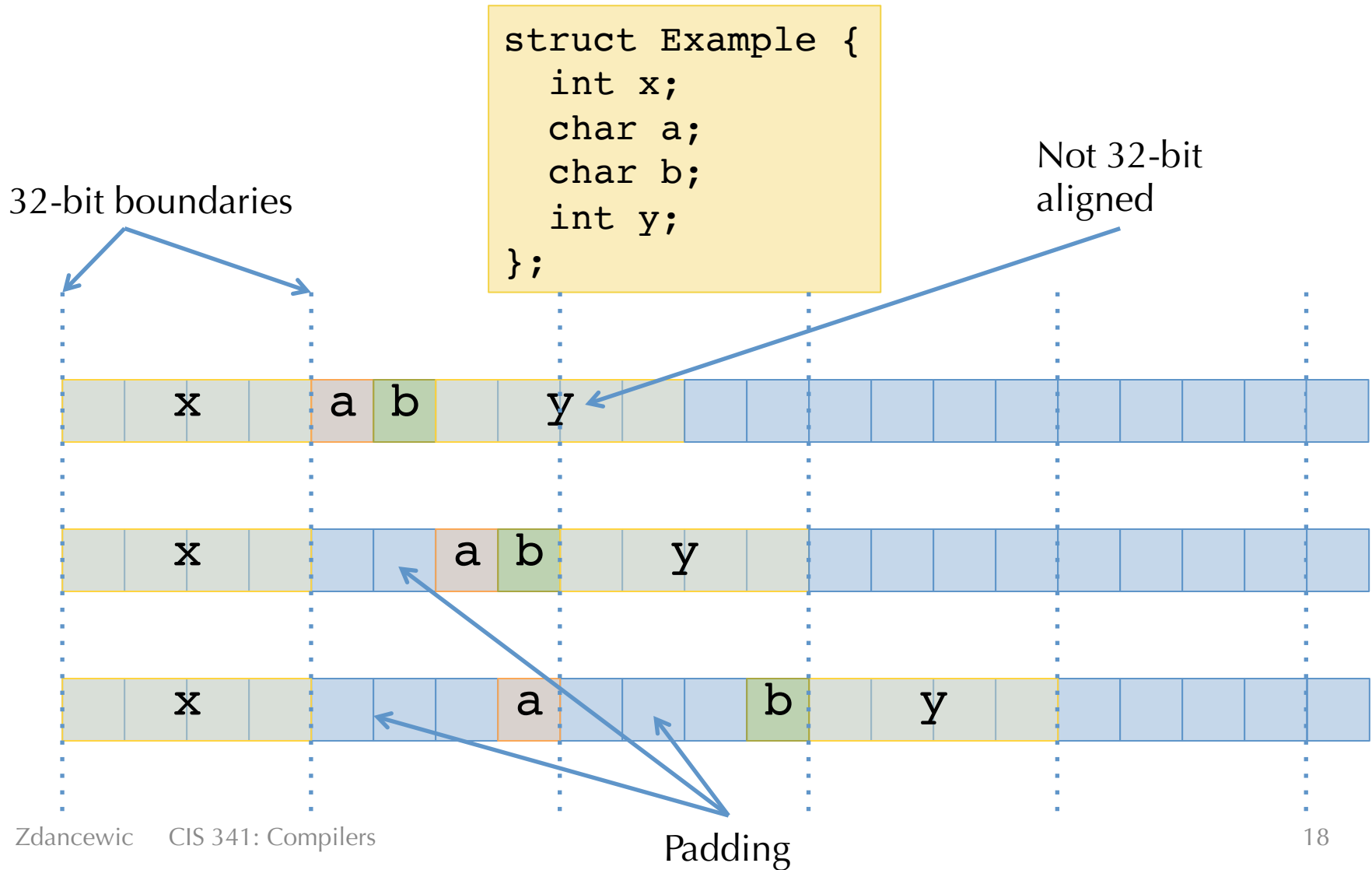


```
struct Point { int x; int y; };  
struct Rect  { struct Point ll, lr, ul, ur };
```

- Consider: `[[square.ul.y]] = (x86.operand, x86.insns)`
- Assume that `%rcx` holds the base address of `square`
- Calculate the offset relative to the base pointer of the data:
 - `ul = sizeof(struct Point) + sizeof(struct Point)`
 - `y = sizeof(int)`
- So: `[[square.ul.y]] = (ans, Movq 20(%rcx) ans)`

Padding & Alignment

- How to lay out non-homogeneous structured data?



Copy-in/Copy-out

When we do an assignment in C as in:

```
struct Rect mk_square(struct Point ll, int elen) {  
    struct Square res;  
    res.lr = ll;  
    ...  
}
```

then we copy all of the elements out of the source and put them in the target. Same as doing word-level operations:

```
struct Rect mk_square(struct Point ll, int elen) {  
    struct Square res;  
    res.lr.x = ll.x;  
    res.lr.y = ll.x;  
    ...  
}
```

- For really large copies, the compiler uses something like `memcpy` (which is implemented using a loop in assembly).

C Procedure Calls

- Similarly, when we call a procedure, we copy arguments in, and copy results out.
 - Caller sets aside extra space in its frame to store results that are bigger than will fit in `%rax`.
 - We do the same with scalar values such as integers or doubles.
- Sometimes, this is termed "call-by-value".
 - This is bad terminology.
 - Copy-in/copy-out is more accurate.
- Benefit: locality
- Problem: expensive for large records...
- In C: can opt to pass *pointers* to structs: "call-by-reference"
- Languages like Java and OCaml always pass non-word-sized objects by reference.

Call-by-Reference:

```
void mkSquare(struct Point *ll, int elen,
             struct Rect *res) {
    res->lr = res->ul = res->ur = res->ll = *ll;
    res->lr.x += elen;
    res->ur.x += elen;
    res->ur.y += elen;
    res->ul.y += elen;
}

void foo() {
    struct Point origin = {0,0};
    struct Square unit_sq;
    mkSquare(&origin, 1, &unit_sq);
}
```

- The caller passes in the address of the point and the address of the result (1 word each).
- Note that returning references to stack-allocated data can cause problems.
 - Need to allocate storage in the heap...



ARRAYS

Arrays

```
void foo() {  
    char buf[27];  
  
    buf[0] = 'a';  
    buf[1] = 'b';  
    ...  
    buf[25] = 'z';  
    buf[26] = 0;  
}
```

```
void foo() {  
    char buf[27];  
  
    *(buf) = 'a';  
    *(buf+1) = 'b';  
    ...  
    *(buf+25) = 'z';  
    *(buf+26) = 0;  
}
```

- Space is allocated on the stack for buf.
 - Note, without the ability to allocated stack space dynamically (C's `alloca` function) need to know size of buf at compile time...
- `buf[i]` is really just: $(\text{base_of_array}) + i * \text{elt_size}$

Multi-Dimensional Arrays

- In C, **int M[4][3]** yields an array with 4 rows and 3 columns.
- Laid out in *row-major* order:

| | | | | | | | |
|---------|---------|---------|---------|---------|---------|---------|-----|
| M[0][0] | M[0][1] | M[0][2] | M[1][0] | M[1][1] | M[1][2] | M[2][0] | ... |
|---------|---------|---------|---------|---------|---------|---------|-----|

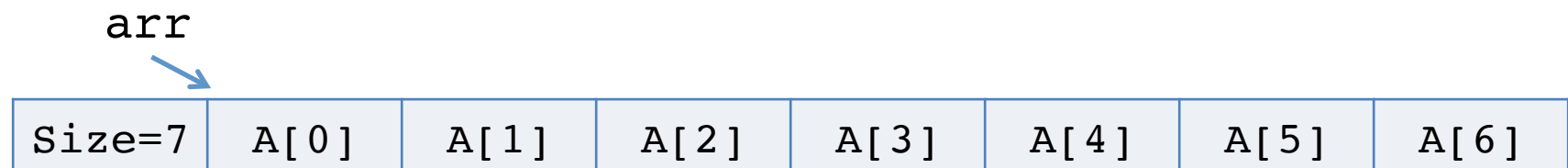
- M[i][j] compiles to?
- In Fortran, arrays are laid out in *column major* order.

| | | | | | | | |
|---------|---------|---------|---------|---------|---------|---------|-----|
| M[0][0] | M[1][0] | M[2][0] | M[3][0] | M[0][1] | M[1][1] | M[2][1] | ... |
|---------|---------|---------|---------|---------|---------|---------|-----|

- In ML and Java, there are no multi-dimensional arrays:
 - (int array) array is represented as an array of pointers to arrays of ints.
- Why is knowing these memory layout strategies important?

Array Bounds Checks

- Safe languages (e.g. Java, C#, ML but not C, C++) check array indices to ensure that they're in bounds.
 - Compiler generates code to test that the computed offset is legal
- Needs to know the size of the array... where to store it?
 - One answer: Store the size *before* the array contents.



- Other possibilities:
 - Pascal: only permit statically known array sizes (very unwieldy in practice)
 - What about multi-dimensional arrays?

Array Bounds Checks (Implementation)

- Example: Assume `%rax` holds the base pointer (`arr`) and `%ecx` holds the array index `i`. To read a value from the array `arr[i]`:

```
    movq -8(%rax) %rdx          // load size into rdx
    cmpq %rdx %rcx              // compare index to bound
    j 1 __ok                    // jump if 0 <= i < size
    callq __err_oob             // test failed, call the error handler
__ok:
    movq (%rax, %rcx, 8) dest    // do the load from the array access
```

- Clearly more expensive: adds move, comparison & jump
 - More memory traffic
 - Hardware can improve performance: executing instructions in parallel, branch prediction
- These overheads are particularly bad in an inner loop
- Compiler optimizations can help remove the overhead
 - e.g. In a for loop, if bound on index is known, only do the test once

C-style Strings

- A string constant "foo" is represented as global data:

```
_string42: 102 111 111 0
```

- C uses null-terminated strings
- Strings are usually placed in the *text* segment so they are read only.
 - allows all copies of the same string to be shared.
- Rookie mistake (in C): write to a string constant.

```
char *p = "foo";  
p[0] = 'b';
```

- Instead, must allocate space on the heap:

```
char *p = (char *)malloc(4 * sizeof(char));  
strncpy(p, "foo", 4);    /* include the null byte */  
p[0] = 'b';
```



TAGGED DATATYPES

C-style Enumerations / ML-style datatypes

- In C:

```
enum Day {sun, mon, tue, wed, thu, fri, sat} today;
```

- In ML:

```
type day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
```

- Associate an integer *tag* with each case: sun = 0, mon = 1, ...
 - C lets programmers choose the tags

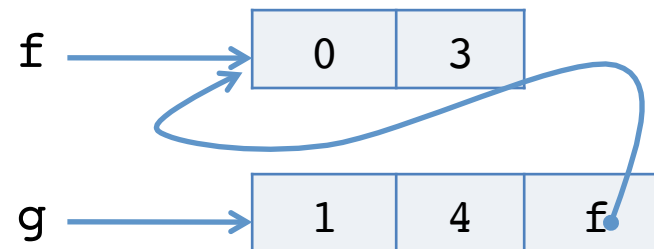
- ML datatypes can also carry data:

```
type foo = Bar of int | Baz of int * foo
```

- Representation: a `foo` value is a pointer to a pair: (tag, data)
- Example: tag(`Bar`) = 0, tag(`Baz`) = 1

`[[let f = Bar(3)]] =`

`[[let g = Baz(4, f)]] =`



Switch Compilation

- Consider the C statement:

```
switch (e) {  
    case sun: s1; break;  
    case mon: s2; break;  
    ...  
    case sat: s3; break;  
}
```

- How to compile this?
 - What happens if some of the break statements are omitted? (Control falls through to the next branch.)

Cascading ifs and Jumps

[[switch(e) {case tag1: s1; case tag2 s2; ...}]] =

- Each \$tag1...\$tagN is just a constant int tag value.
- Note: [[break;]] (within the switch branches) is:
br %merge

```
%tag = [[e]];
br label %l1
l1: %cmp1 = icmp eq %tag, $tag1
    br %cmp1 label %b1, label %merge
b1: [[s1]]
    br label %l2

l2: %cmp2 = icmp eq %tag, $tag2
    br %cmp2 label %b2, label %merge
b2: [[s2]]
    br label %l3

...
lN: %cmpN = icmp eq %tag, $tagN
    br %cmpN label %bN, label %merge
bN: [[sN]]
    br label %merge

merge:
```

Alternatives for Switch Compilation

- Nested if-then-else works OK in practice if # of branches is small
 - (e.g. < 16 or so).
- For more branches, use better datastructures to organize the jumps:
 - Create a table of pairs (v1, branch_label) and loop through
 - Or, do binary search rather than linear search
 - Or, use a hash table rather than binary search
- One common case: the tags are dense in some range [min...max]
 - Let $N = \text{max} - \text{min}$
 - Create a branch table `Branches[N]` where `Branches[i] = branch_label` for tag `i`.
 - Compute `tag = ⌊e⌋` and then do an *indirect jump*: `J Branches[tag]`
- Common to use heuristics to combine these techniques.

ML-style Pattern Matching

- ML-style match statements are like C's switch statements except:
 - Patterns can bind variables
 - Patterns can nest

```
match e with
| Bar(z) -> e1
| Baz(y, Bar(w)) -> e2
| _ -> e3
```



```
match e with
| Bar(z) -> e1
| Baz(y, tmp) ->
    (match tmp with
     | Bar(w) -> e2
     | Baz(_, _) -> e3)
```

- Compilation strategy:
 - “Flatten” nested patterns into matches against one constructor at a time.
 - Compile the match against the tags of the datatype as for C-style switches.
 - Code for each branch additionally must copy data from $\llbracket e \rrbracket$ to the variables bound in the patterns.
- There are many opportunities for optimization, many papers about “pattern-match compilation”
 - Many of these transformations can be done at the AST level