YSC4230: Programming Language Design and Implementation

Week 4: Simple IRs and LLVM

Ilya Sergey

ilya.sergey@yale-nus.edu.sg

Intermediate Representations

Why do something else?

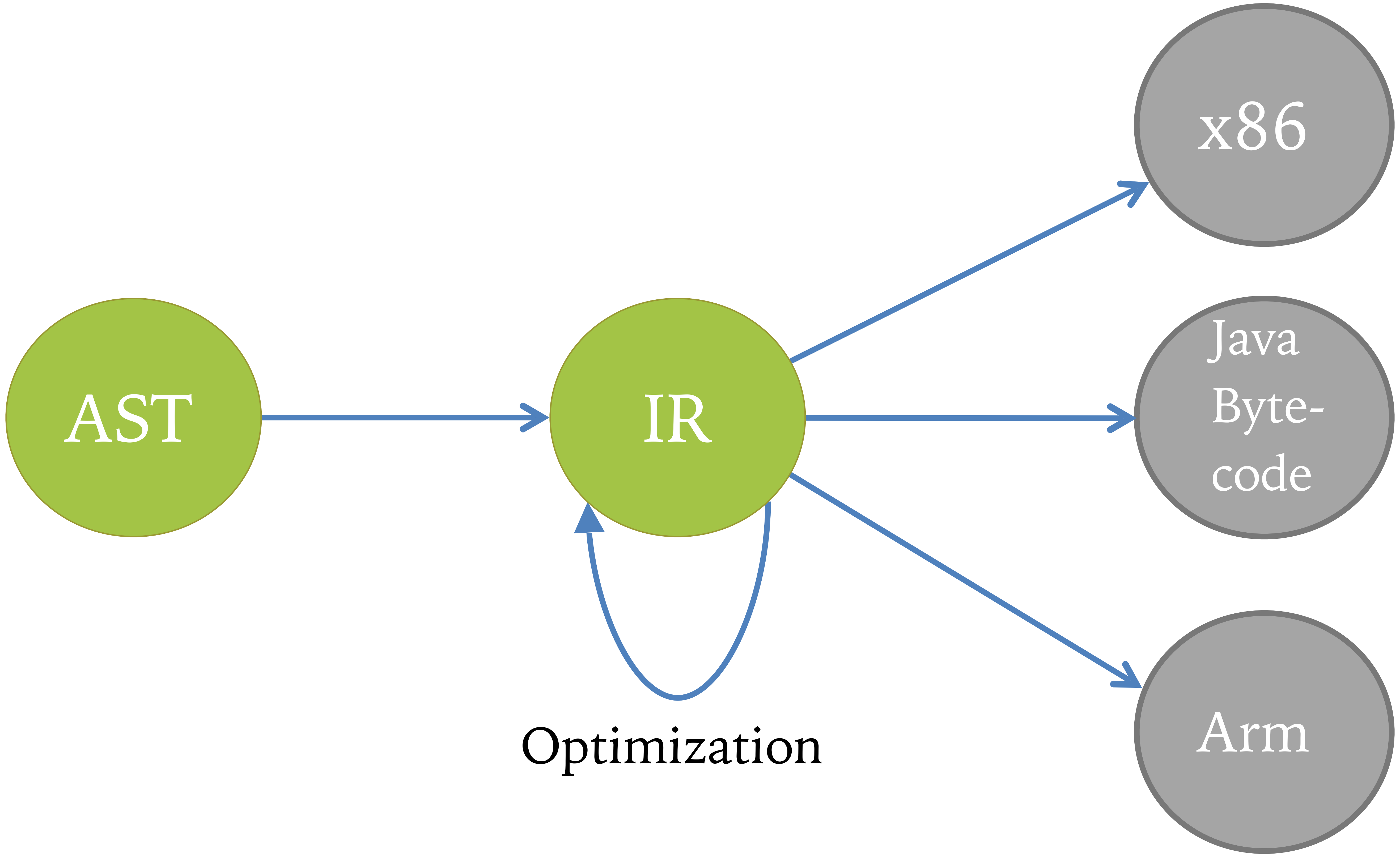
- We have seen a simple *syntax-directed* translation
  - Input syntax uniquely determines the output, no complex analysis or code transformation is done.
  - It works fine for simple languages.

But…

- The resulting code quality is poor.
- Richer source language features are hard to encode
  - Structured data types, objects, first-class functions, etc.
- It's hard to optimize the resulting assembly code.
  - The representation is too concrete – e.g. it has committed to using certain registers and the stack
  - Only a fixed number of registers
  - Some instructions have restrictions on where the operands are located
- Control-flow is not structured:
  - Arbitrary jumps from one code block to another
  - Implicit fall-through makes sequences of code non-modular (i.e. you can't rearrange sequences of code easily)
- Retargeting the compiler to a new architecture is hard.
  - Target assembly code is hard-wired into the translation

Intermediate Representations (IR's)

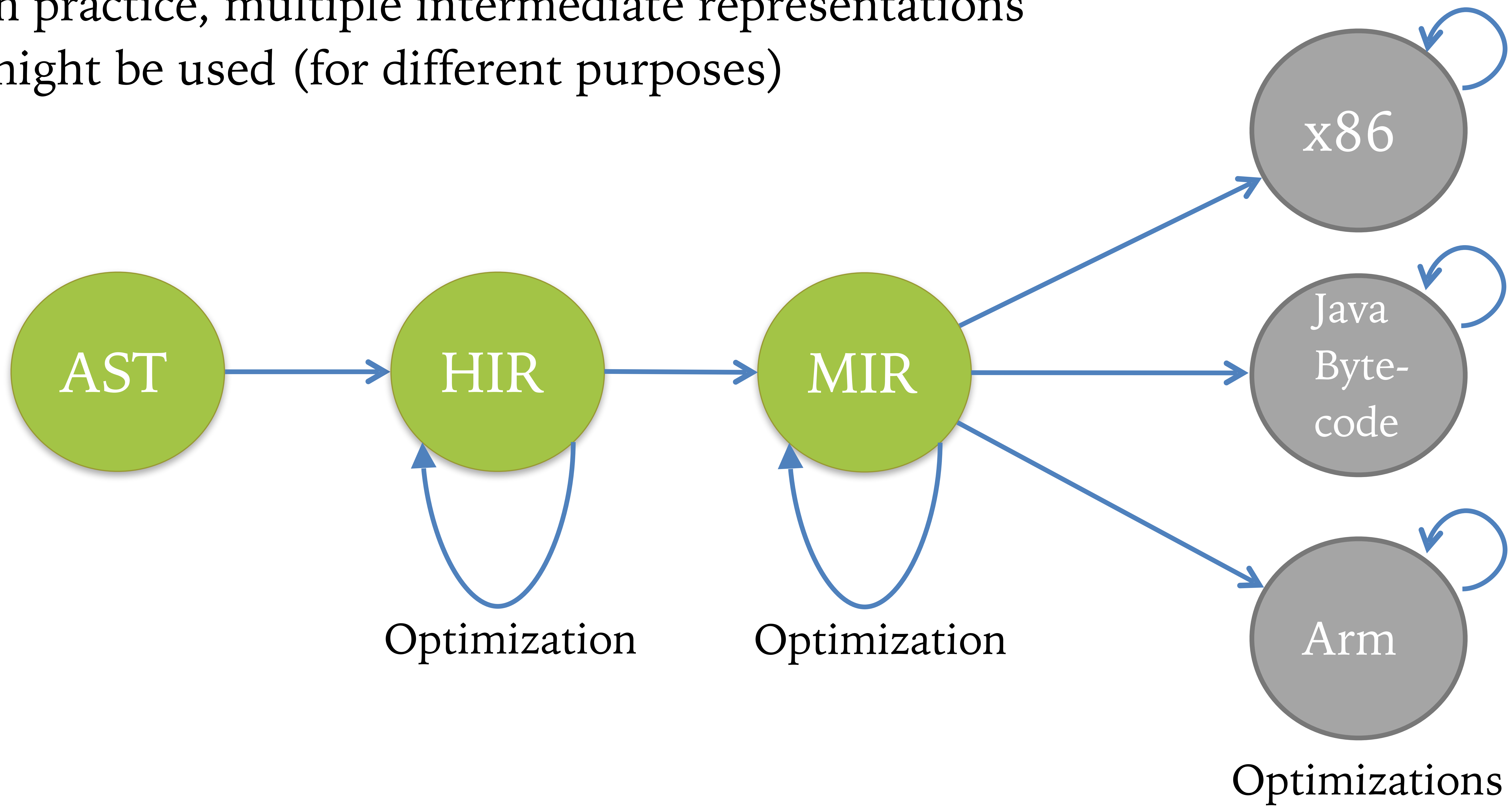
- Abstract machine code: hides details of the target architecture
- Allows machine independent code generation and optimization.



Multiple IR's

- Goal: get program closer to machine code without losing the information needed to do analysis and optimizations

- In practice, multiple intermediate representations might be used (for different purposes)



What makes a good IR?

- Easy translation target (from the level above)
- Easy to translate (to the level below)
- Narrow interface
  - Fewer constructs means simpler phases/optimizations

- Example: Source language might have “while”, “for”, and “foreach” loops (and maybe more variants)
  - IR might have only “while” loops and sequencing
  - Translation eliminates “for” and “foreach”

```
⟦for(pre; cond; post) {body}⟧
 =
  ⟦pre; while(cond) {body;post}⟧
```

  - Here the notation ⟦cmd⟧ denotes the “translation” or “compilation” of the command cmd.

IR's at the extreme

- High-level IR's
  - Abstract syntax + new node types not generated by the parser
    - e.g. Type checking information or disambiguated syntax nodes
  - Typically preserves the high-level language constructs
    - Structured control flow, variable names, methods, functions, etc.
    - May do some simplification (e.g. convert `for` to `while`)
  - Allows high-level optimizations based on program structure
    - e.g. inlining "small" functions, reuse of constants, etc.
  - Useful for semantic analyses like type checking

- Low-level IR's
  - Machine dependent assembly code + extra pseudo-instructions
    - e.g. a pseudo instruction for interfacing with garbage collector or memory allocator (parts of the language runtime system)
    - e.g. (on x86) a `imulq` instruction that doesn't restrict register usage
  - Source structure of the program is lost:
    - Translation to assembly code is straightforward
  - Allows low-level optimizations based on target architecture
    - e.g. register allocation, instruction selection, memory layout, etc.

- What's in between?

Mid-level IR's: Many Varieties

- Intermediate between AST (abstract syntax) and assembly
- May have unstructured jumps, abstract registers, or memory locations
- Convenient for translation to high-quality machine code
  - Example: all intermediate values are named to facilitate optimizations that attempt to minimize stack/register usage
- Many examples:
  - Triples: OP a b
    - Useful for instruction selection on X86 via "graph tiling" (a way to better utilise registers)
  - Quadruples: a = b OP c (RISC-like "three address form")
  - SSA: variant of quadruples where each variable is assigned exactly once
    - Easy dataflow analysis for optimization
    - e.g. LLVM: industrial-strength IR, based on SSA
  - Stack-based:
    - Easy to generate
    - e.g. Java Bytecode, UCODE

Growing an IR

- Develop an IR in detail… starting from the very basic.
- Start: a (very) simple intermediate representation for the *arithmetic language*
  - Very high level
  - No control flow
- Goal: A simple subset of the LLVM IR
  - LLVM = "Low-level Virtual Machine"
  - Used in HW3+
- Add features needed to compile rich source languages

Simple let-based IR

Eliminating Nested Expressions

- Fundamental problem:
  - Compiling complex & nested expression forms to simple operations.

Source

```
((1 + X4) + (3 + (X1 * 5)))
```

AST

```
Add(Add(Const 1, Var X4),
    Add(Const 3, Mul(Var X1,
                     Const 5)))
```

IR

?

- Idea: *name* intermediate values, make order of evaluation explicit.
  - No nested operations.

Translation to SLL

- Given this:

```
Add(Add(Const 1, Var X4),
    Add(Const 3, Mul(Var X1,
                     Const 5)))
```

- Translate to this desired SLL form:

```
let tmp0 = add 1L varX4 in
let tmp1 = mul varX1 5L in
let tmp2 = add 3L tmp1 in
let tmp3 = add tmp0 tmp2 in
tmp3
```

- Translation makes the order of evaluation explicit.
- Names intermediate values
- Note: introduced temporaries are never modified

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

Demo: IR1 and IR2

- https://github.com/ysc4230/week-03-intermediate-2021
- Definitions: `ir1.ml, ir2.ml`
- Using IRs: `ir_by_hand.ml`

IR3: 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.

Demo: IR3

- https://github.com/ysc4230/week-03-intermediate-2021
- Definitions: `ir3.ml`

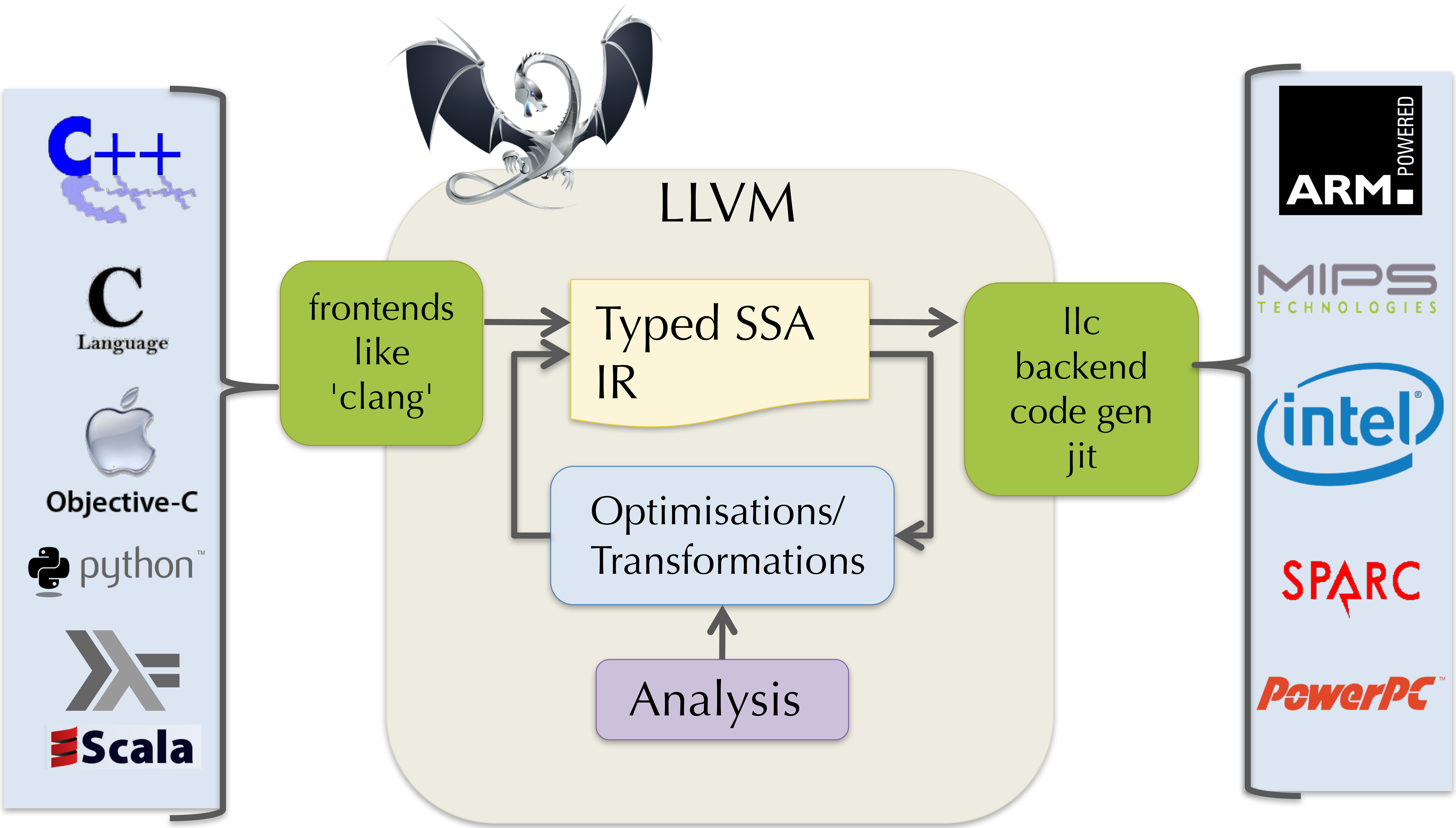
LLVM

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 Multi-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.

LLVM Compiler Infrastructure

[Lattner et al.]



Example LLVM Code

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

factorial64.c

```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

```llvm
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

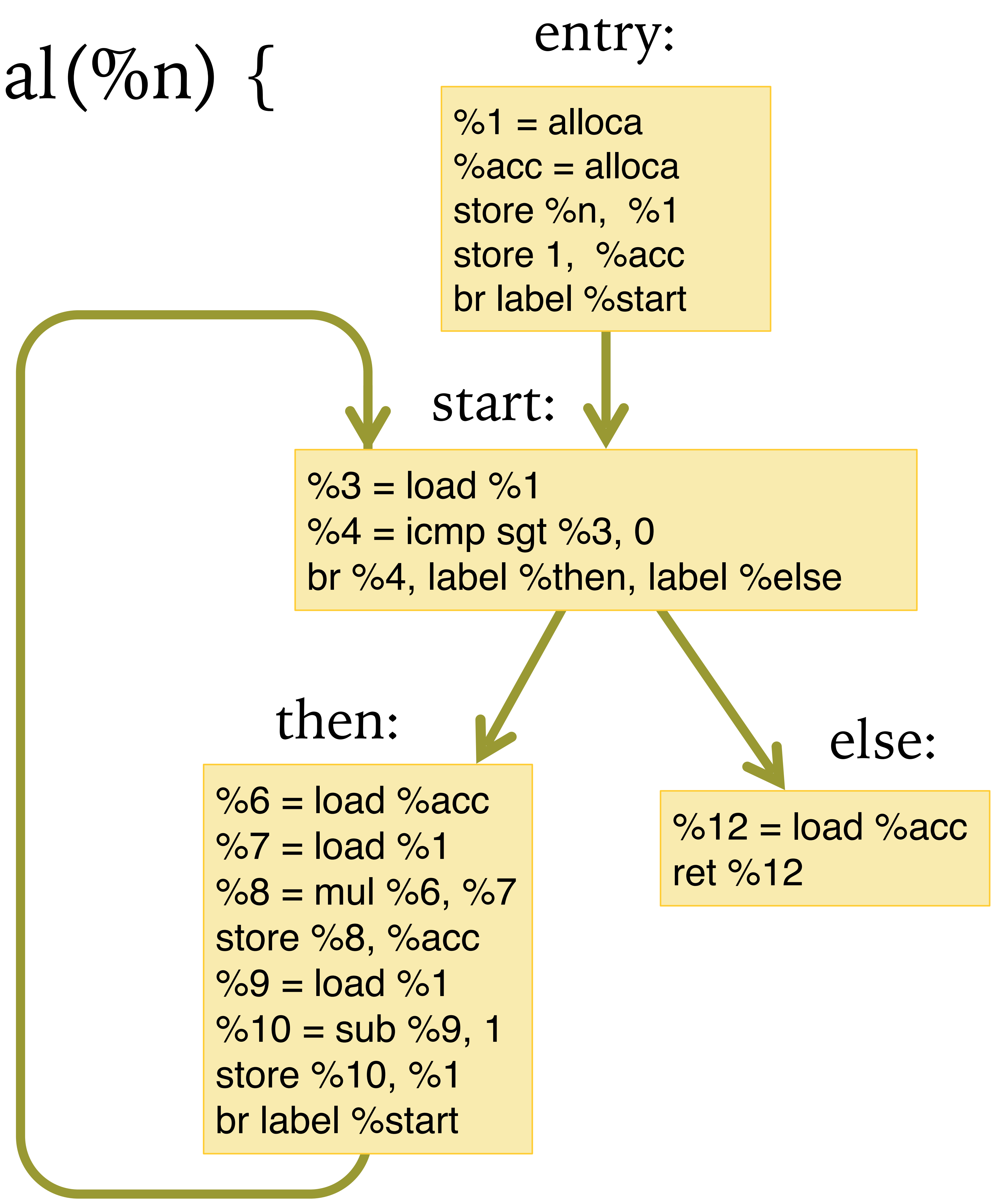
- Decorates values with type information
  i64
  i64*
  i1 (boolean)
- Permits numeric identifiers
- Has alignment annotations
  (padding for some specified number of bytes)
- Keeps track of entry edges for each block:
  preds = %5, %0

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) {



}

```
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
}
```

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, unlabelled, 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 4230, i64* %acc.       ; store the integer value 4230
%x = load i64, i64* %acc         ; load the value 4230 into %x
```

- Gives an abstract version of stack slots

Structured Data

Compiling Structured Data

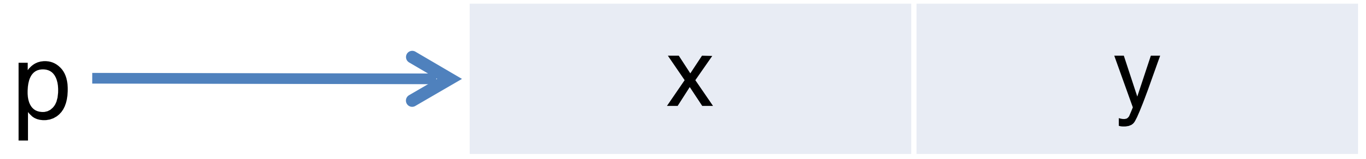
- Consider C-style structures like those below.
- How do we represent Point and Rect values?

```
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

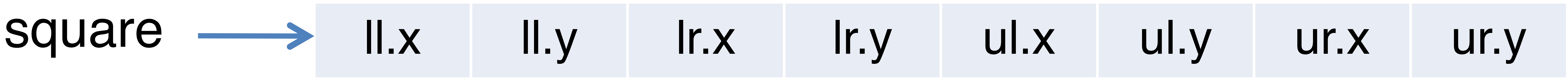
struct Point { int x; int y;};

- 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

square →

ll.x	ll.y	lr.x	lr.y	ul.x	ul.y	ur.x	ur.y

```
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

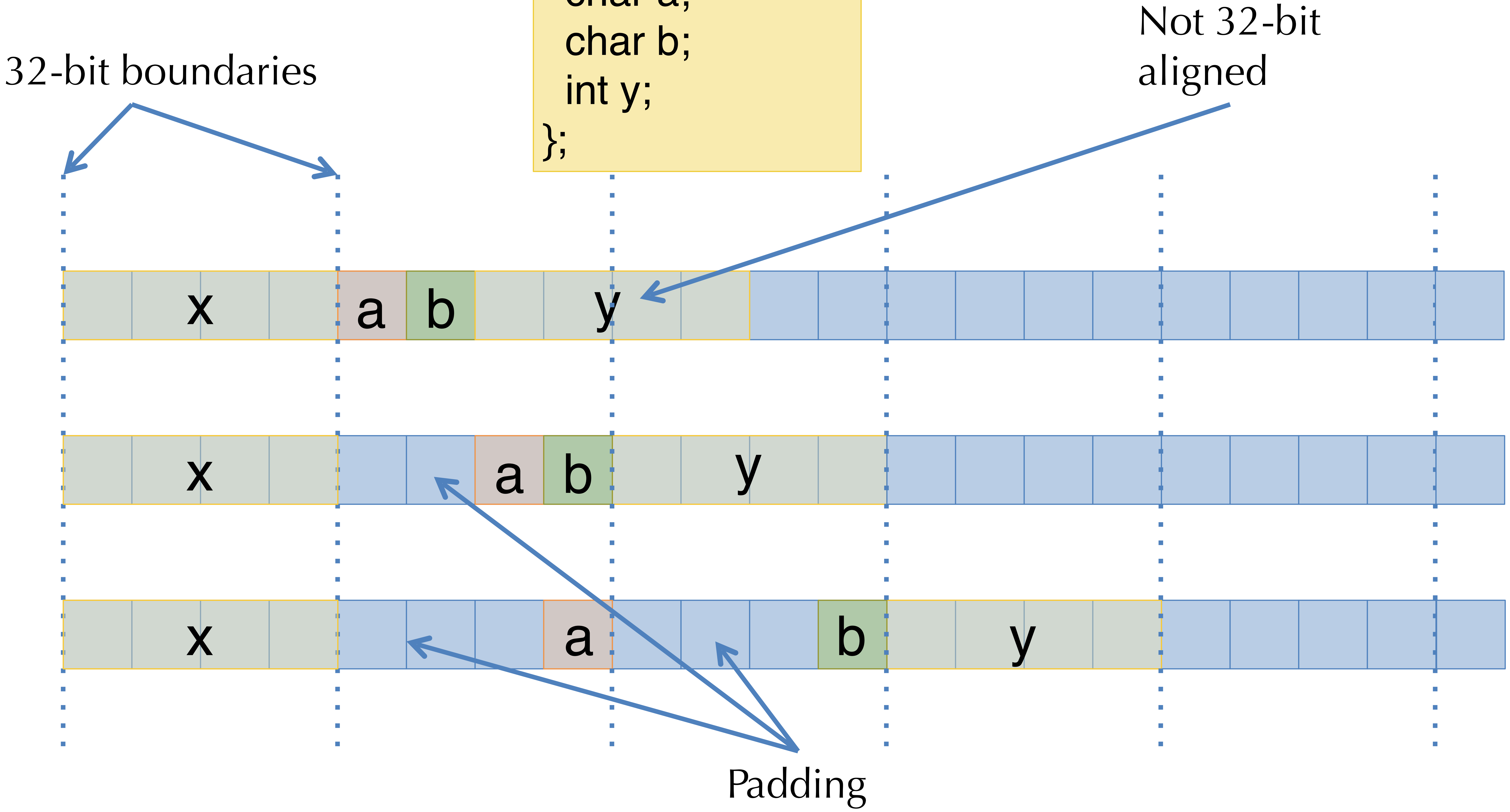
- Calculate the offset relative to the base pointer of the data:
  - ul = sizeof(struct Point) + sizeof(struct Point)
  - y = sizeof(int)

- So: ⟦square.ul.y⟧ = (ans, Movq 20(%rcx) ans)

Padding & Alignment

- How to lay out non-homogeneous structured data?

```
struct Example {
  int x;
  char a;
  char b;
  int y;
};
```



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.
  - This space might be reclaimed when foo() is done
  - Need to allocate storage in the heap…

Working with 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
  (base_of_array) + i * 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]	…

- In Fortran, arrays are laid out in *column major order*.

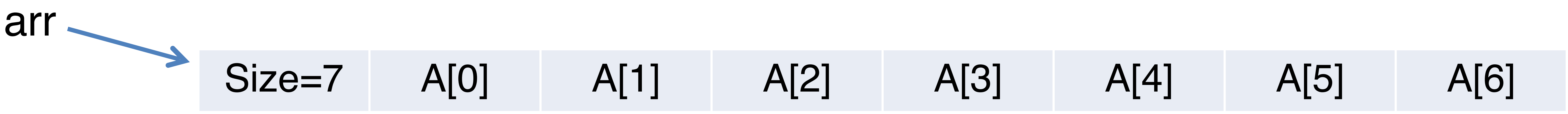
M[0][0]	M[1][0]	M[2][0]	M[3][0]	M[0][1]	M[1][1]	M[2][1]	…

- 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 l __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 optimisations 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’;
```

  Attempting to modify the string literal is *undefined behaviour.*

- 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’;
```

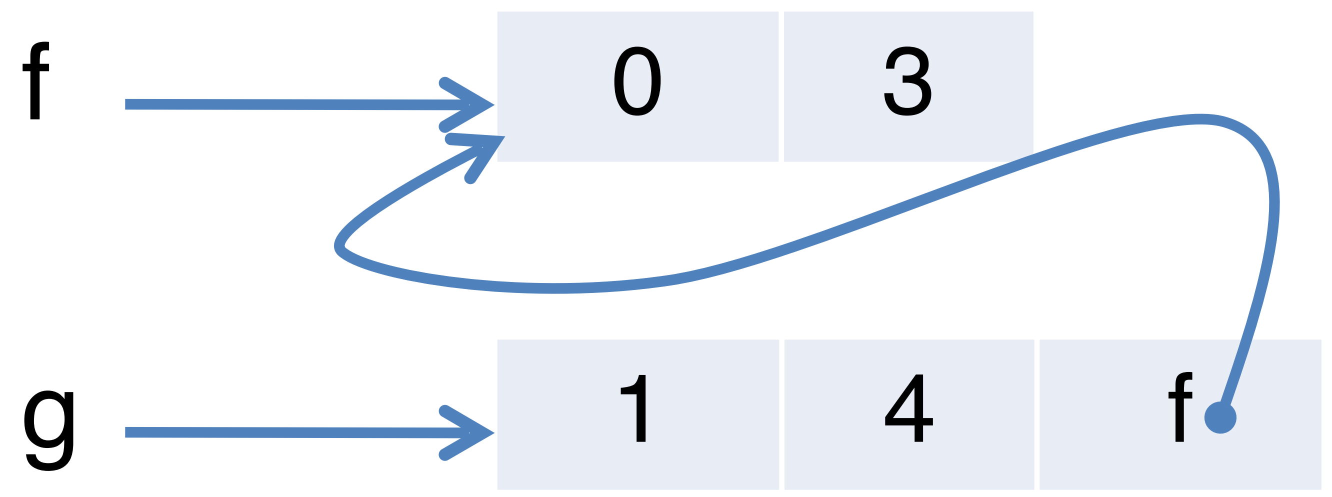
Tagged Datatypes

C-style Enumerations / ML-style datatypes

- In C: `enum Day {sun, mon, tue, wed, thu, fri, sat} today;`
- In OCaml: `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
- OCaml 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 %l2
b1: ⟦s1⟧
        br label %l2

l2: %cmp2 = icmp eq %tag, $tag2
        br %cmp2 label %b2, label %l3
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 data structures to organise 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 = max – 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 ⟦e⟧ to the variables bound in the patterns.
- There are many opportunities for optimisations, many papers about "pattern-match compilation"
  - Many of these transformations can be done at the AST level

Datatypes in LLVM IR

Structured Data in LLVM

- LLVM's IR is uses types to describe the structure of data.

```
t ::=
    void
    i1 | i8 | i64              N-bit integers
    [<#elts> x t]              arrays
    fty                        function types
    {t1, t2, … , tn}           structures
    t*                         pointers
    %Tident                    named (identified) type

fty ::=            Function Types
    t (t1, .., tn)             return, argument types
```

- <#elts> is an integer constant >= 0
- Structure types can be named at the top level:

```
%T1 = type {t1, t2, … , tn}
```

- Such structure types can be recursive

Example LL Types

- A static array of `4230` integers: `[ 4230 x i64 ]`

- A two-dimensional array of integers: `[ 3 x [ 4 x i64 ] ]`

- Structure for representing dynamically-allocated arrays with their length:

```
{ i64 , [0 x i64] }
```

  - There is no array-bounds check; the static type information is only used for calculating pointer offsets.

- C-style linked lists (declared at the top level):

```
%Node = type { i64, %Node*}
```

- Structs from the C program shown earlier:

```
%Rect = { %Point, %Point, %Point, %Point }
%Point = { i64, i64 }
```

getelementptr

- LLVM provides the `getelementptr` instruction to compute pointer values
  - Given a pointer and a "path" through the structured data pointed to by that pointer, `getelementptr` computes an address
  - This is the abstract analog of the X86 LEA (load effective address). It **does not** access memory.
  - It is a "type indexed" operation, since the size computations depend on the type

```
insn ::= …
      |  getelementptr t* %val, t1 idx1, t2 idx2 ,…
```

- Example: access the x component of the first point of a rectangle:

```
%tmp1 = getelementptr %Rect* %square, i32 0, i32 0
%tmp2 = getelementptr %Point* %tmp1, i32 0, i32 0
```

- The first is **i32 0** a "step through" the pointer to, e.g., %square, with offset 0.

See "Why is the extra 0 index required?": https://llvm.org/docs/GetElementPtr.html#why-is-the-extra-0-index-required

GEP Example*

```
struct RT {
     int A;
     int B[10][20];
     int C;
}
struct ST {
     struct RT X;
     int Y;
     struct RT Z;
}
int *foo(struct ST *s) {
  return &s[1].Z.B[5][13];
}
```

1. `%s` is a pointer to an (array of) `%ST` structs, suppose the pointer value is ADDR
2. Compute the index of the 1st element by adding `size_ty(%ST)`.
3. Compute the index of the `Z` field by adding `size_ty(%RT) + size_ty(i32)` to skip past `X` and `Y`.
4. Compute the index of the `B` field by adding `size_ty(i32)` to skip past `A`.
5. Index into the 2d array.

```
%RT = type { i32, [10 x [20 x i32]], i32 }
%ST = type { %RT, i32, %RT }
define i32* @foo(%ST* %s) {
entry:
    %arrayidx = getelementptr %ST* %s, i32 1, i32 2, i32 1, i32 5, i32 13
    ret i32* %arrayidx
}
```

Final answer: ADDR + `size_ty(%ST) + size_ty(%RT) + size_ty(i32)`
`+ size_ty(i32) + 5*20*size_ty(i32) + 13*size_ty(i32)`

*adapted from the LLVM documentation: see http://llvm.org/docs/LangRef.html#getelementptr-instruction

getelementptr

- GEP *never* dereferences the address it's calculating:
  - GEP only produces pointers by doing arithmetic
  - It doesn't actually traverse the links of a data structure

- To index into a deeply nested structure, one has to "follow the pointer" by loading from the computed pointer

Compiling Data Structures via LLVM

1. Translate high level language types into an LLVM representation type.
   - For some languages (e.g. C) this process is straight forward
     - The translation simply uses platform-specific alignment and padding
   - For other languages, (e.g. OO languages) there might be a fairly complex elaboration.
     - e.g. for OCaml, arrays types might be translated to pointers to length-indexed structs.
       `⟦int array⟧ = { i32, [0 x i32]}*`
2. Translate accesses of the data into getelementptr operations:
   - e.g. for OCaml array size access:
     ⟦length a⟧ =
     `%1 = getelementptr {i32, [0 x i32]}* %a, i32 0, i32 0`

Type Casting

- What if the LLVM IR's type system isn't expressive enough?
  - e.g. if the source language has subtyping, perhaps due to inheritance
  - e.g. if the source language has polymorphic/generic types

- LLVM IR provides a `bitcast` instruction
  - This is a form of (potentially) unsafe cast. Misuse can cause serious bugs (segmentation faults, or silent memory corruption)

```
%rect2 = type { i64, i64 }            ; two-field record
%rect3 = type { i64, i64, i64 }       ; three-field record

define @foo() {
  %1 = alloca %rect3      ; allocate a three-field record
  %2 = bitcast %rect3* %1 to %rect2*     ; safe cast
  %3 = getelementptr %rect2* %2, i32 0, i32 1   ; allowed
  …
}
```

Demo: Compiling to LLVM

- Clone https://github.com/ysc3208/week-04-llvm-demo
- Check `struct.c` and its LLVM representations

Next Week

- LLVMLite Specification
- Overview of HW3
- Lexical Analysis