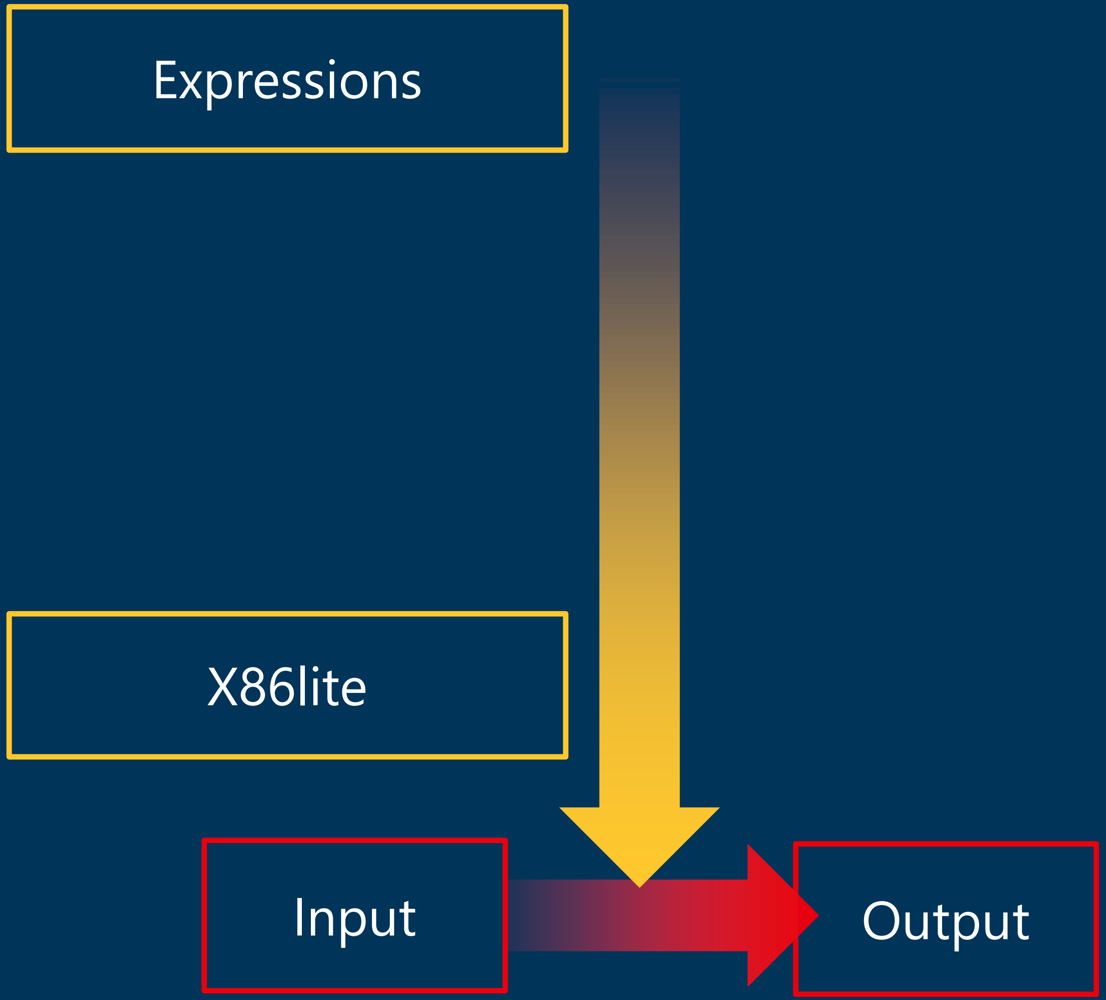
Compiler Construction: LLVMlite

Direct compilation

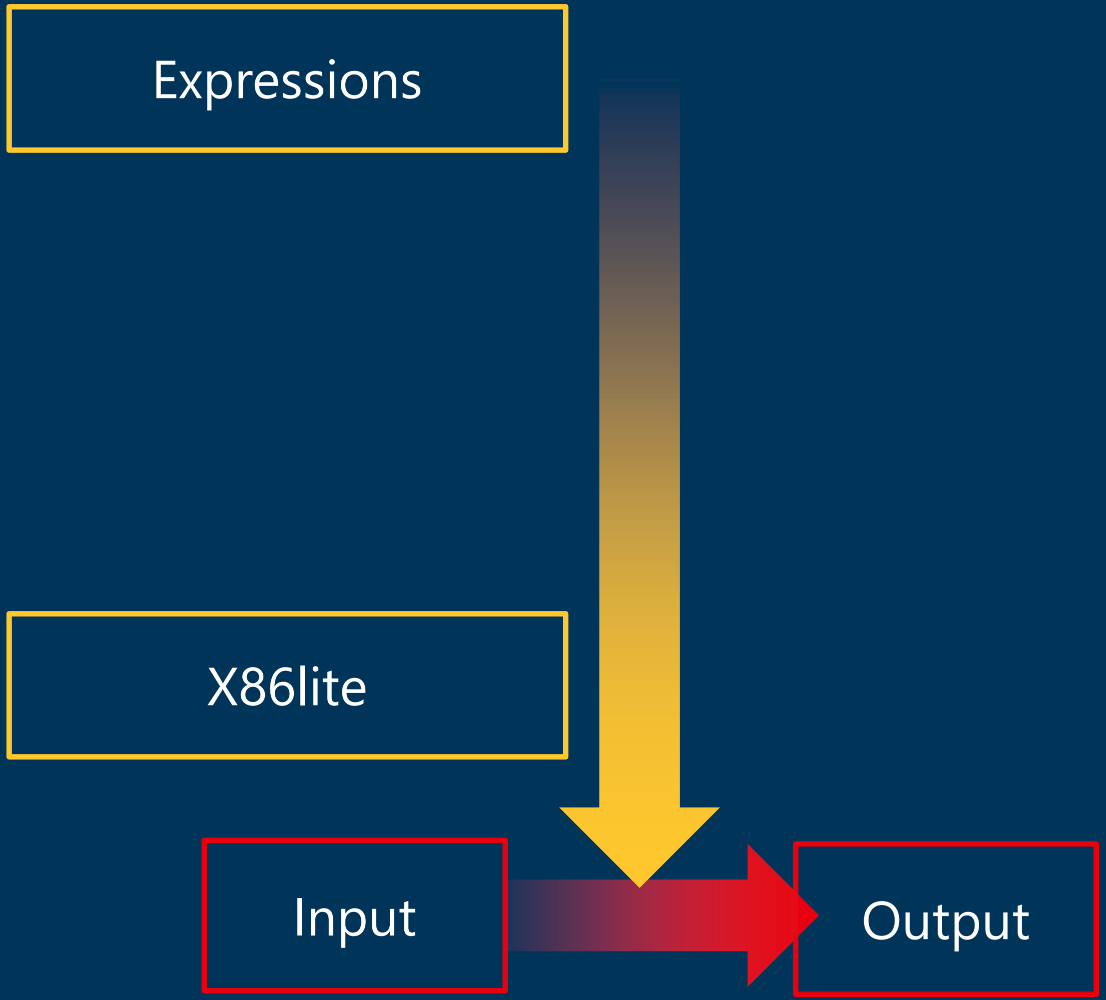


Compile directly from expression language to x86

- *Syntax-directed* compilation scheme
  - Special cases can improve generated code
- *Peephole optimization* of the generated assembly

So, why not do this in general?

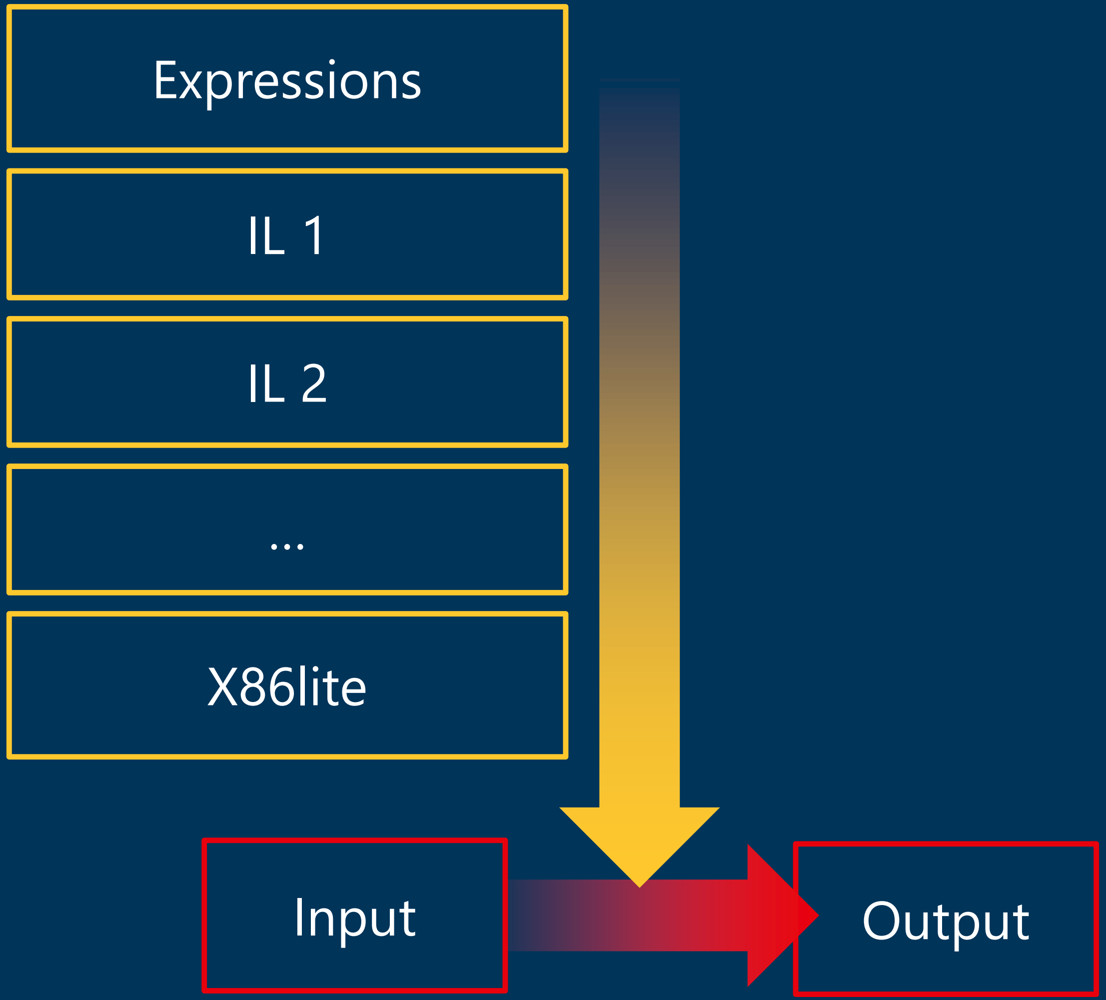
Direct compilation



So, why not do this in general?

- Generated code quality is poor
  - Particularly with non-local properties, like register usage
- More expressive language features difficult to implement
  - Structured data
  - Control-flow structures
  - Objects/first-class functions

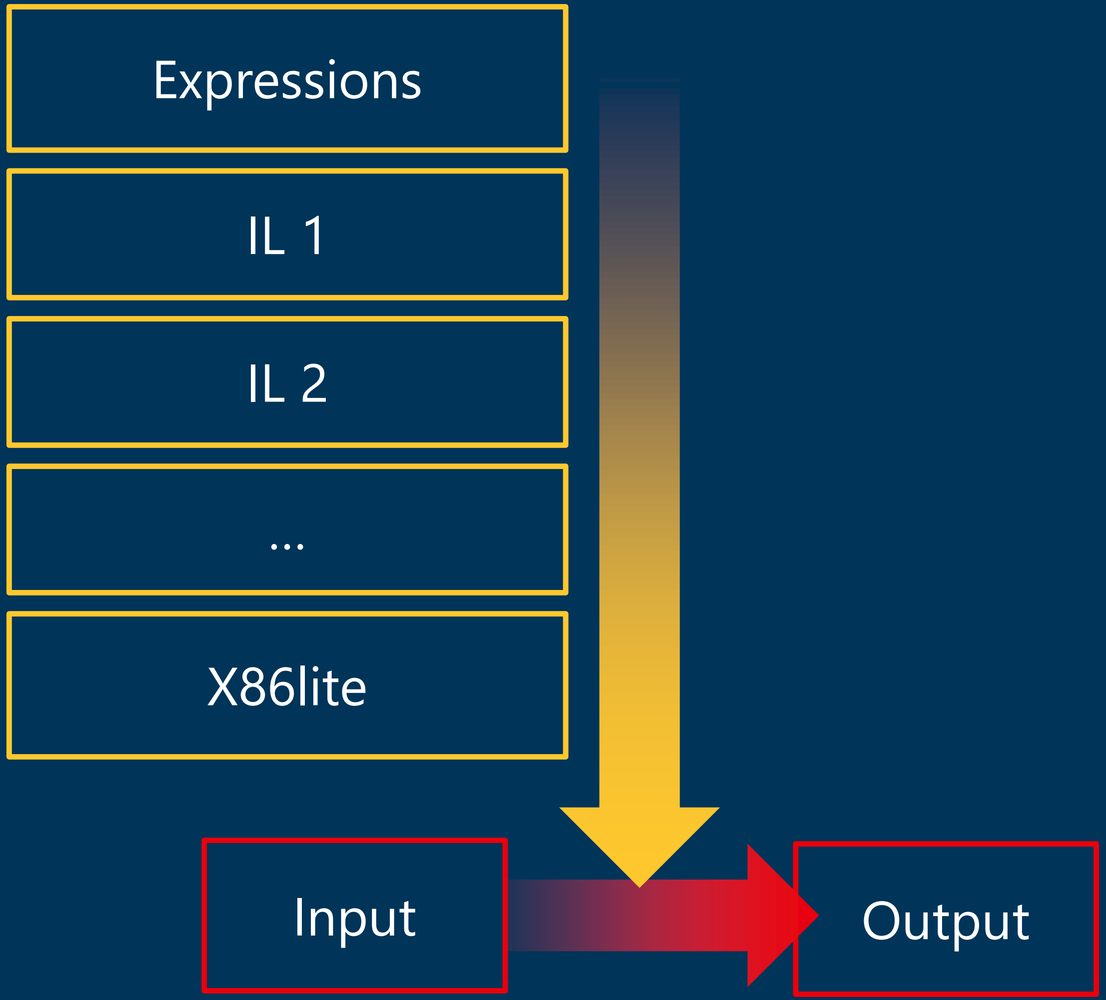
Intermediate languages



What is an intermediate language?

- Reasonable translation target for previous language
- Reasonable translation source for next language
- Enables / simplifies particular optimizations

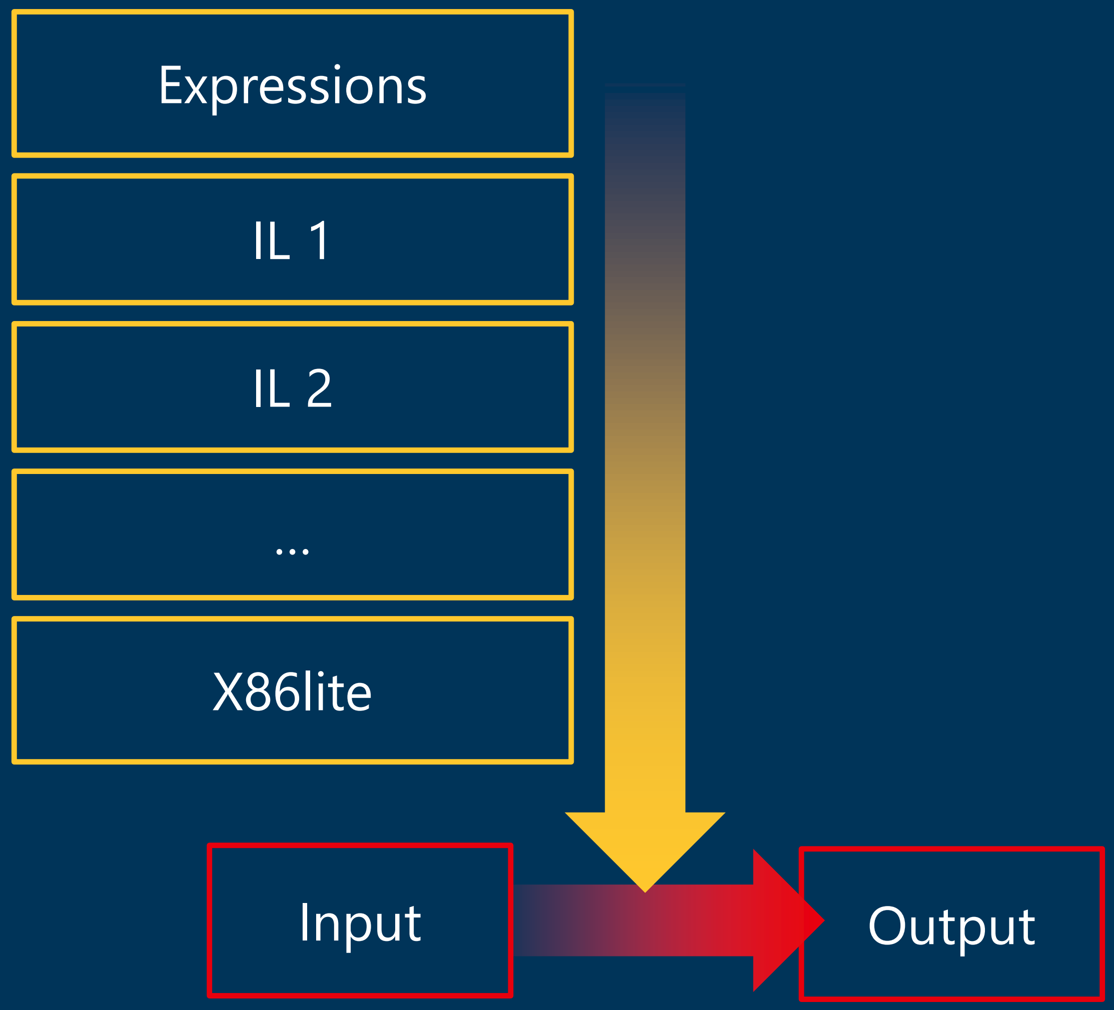
Intermediate languages



Kinds of intermediate languages

- High-level ILs
  - Introduces information not (explicit) in early stages
  - Preserves (but may simplify) high-level structures
  - Function inlining, constant propagation
- Low-level ILs

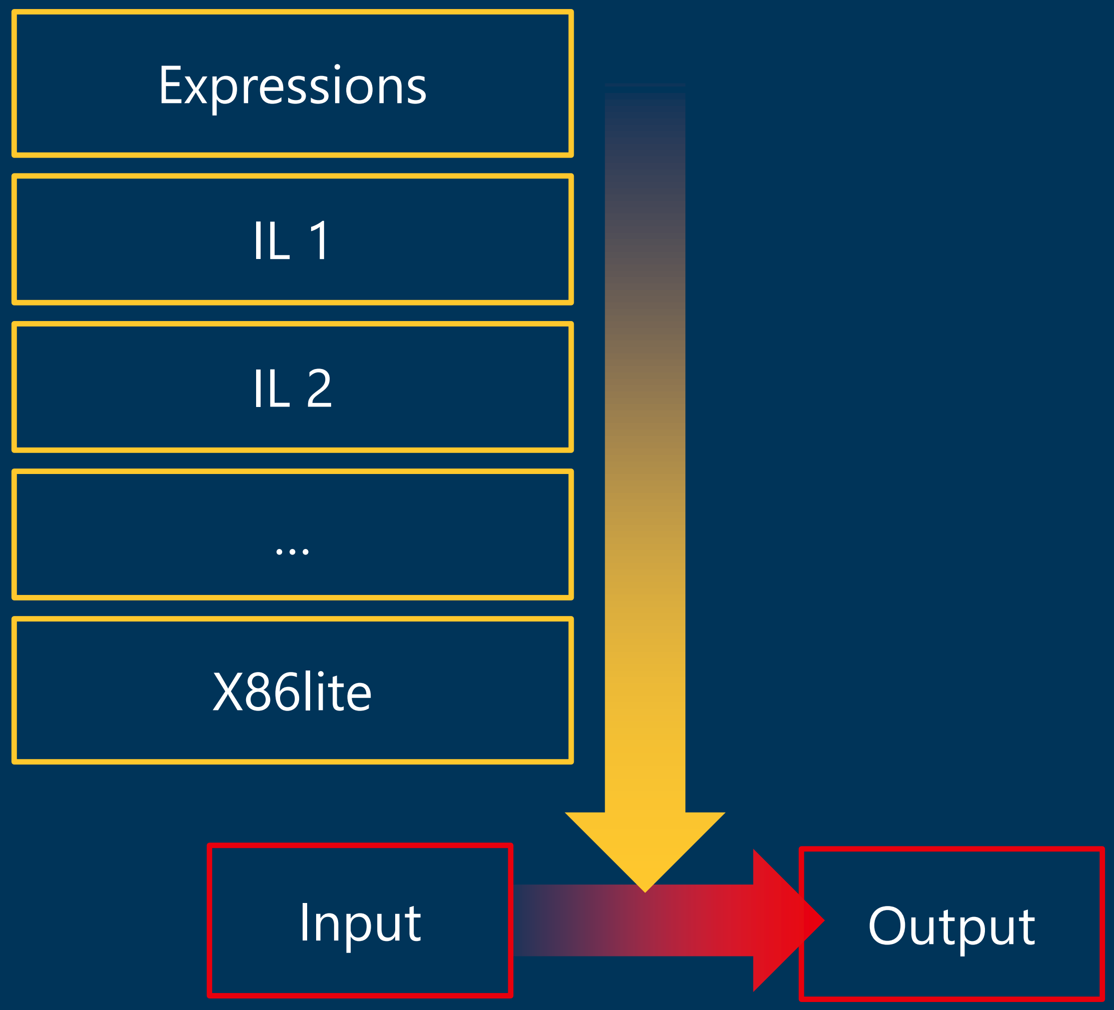
Intermediate languages



Kinds of intermediate languages

- High-level ILs
- Low-level ILs
  - Extensions of assembly code (e.g., pseudo ops for interaction with allocator)
  - Lost structure of source language
  - Register allocation, instruction selection

Intermediate languages



Kinds of intermediate languages

- High-level ILs
- Low-level ILs
- Mid-level ILs
  - Machine-independent, but otherwise low-level
  - Abstractions of memory, flow-of-control

Structuring intermediate languages

- Triples
  - *OP a b* — like (much) x86 assembly
  - Useful for instruction selection
- Stacks
- Three-address form

Structuring intermediate languages

- Triples
- Stacks
  - Instructions all implicitly manipulate stack—*iload_1, iadd*
  - Easy to generate, reasonable to target numerous architectures
  - Very common in practical VMs: JBC, CIL, WebAssembly, CPython, YARV
- Three-address form

Structuring intermediate languages

- Triples
- Stacks
- Three-address form
  - *a = b OP c*
  - Common variant: *single static assignment*
  - Supports easy data-flow and control-flow analysis
  - Very common in practical compilers: GCC, LLVM, MSVC, HotSpot JIT, …

We're going to study SSA intermediate languages

Developing our IL

- Start: simple IL for arithmetic language
  - Codify the invariants used in compiling arithmetic
  - Relatively high level (but still SSA)
  - No control flow
- First goal: subset of LLVM
  - Control flow
  - Reasonable register allocation
- Then: add support for expressive source language features
  - Structured data
  - Closures…

SSA IL for arithmetic

Goal: un-nest nested expressions

```
2 + ((5 + 3) * 5)
```

```
z1 = 5 + 3
z2 = z1 * 5
z3 = z2 + 2
return z3
```

CONTROL FLOW GRAPHS

Basic blocks

Control flow expressed with “tamed” goto:

- Code divided into basic blocks
- Basic blocks arranged into control-flow graph (CFG)

Basic blocks

Control flow expressed with “tamed” goto:

- Code divided into basic blocks
  - Starts with a label (entry point)
  - Ends with a control-flow instruction (branch or return)
  - Contains no other control-flow instructions
  - Contains no interior labels
- Basic blocks arranged into control-flow graph (CFG)

Basic blocks

Control flow expressed with "tamed" goto:

- Code divided into basic blocks
- Basic blocks arranged into control-flow graph (CFG)
  - Basic blocks are the "nodes" of the graph
  - Edge from block *A* to block *B* if the control flow instruction (terminator) at the end of block *A* can jump to block *B*

CFG example

```
define factorial(n) {
start:
  z0 = alloc
  z1 = alloc
  store n, z0
  store 1, z1
  branch loop
loop:
  z3 = load z0
  z4 = z0 > 0
  branch z4, then, else
then:
  z5 = load z1
  z6 = load z0
  z7 = z5 * z6
  store z7, z1
  z8 = z6 - 1
  store z8, z0
  branch loop
else:
  z9 = load z1
  return z9
}
```

CFG example

```
define factorial(n) {

z0 = alloc
z1 = alloc
store n, z0
store 1, z1
branch loop

z3 = load z0
z4 = z0 > 0
branch z4, then, else

z5 = load z1
z6 = load z0
z7 = z5 * z6
store z7, z1
z8 = z6 - 1
store z8, z0
branch loop

z9 = load z1
return z9

}
```

CFGs formally

A CFG is a list of labeled (basic) blocks such that:

- No two blocks have the same label
- The terminator in each block mentions only labels defined in the CFG
- There is a distinguished, unlabeled entry block

LLVM

LLVM (Low-level virtual machine)

What is LLVM?

- Open source compiler infrastructure
- Initially developed by Chris Lattner at UIUC; now primarily supported by Apple.
- Front-ends for C, C++, various niche languages
- Back-ends for x86, Arm, various niche platforms
- Widely used in academic projects

LLVM architecture

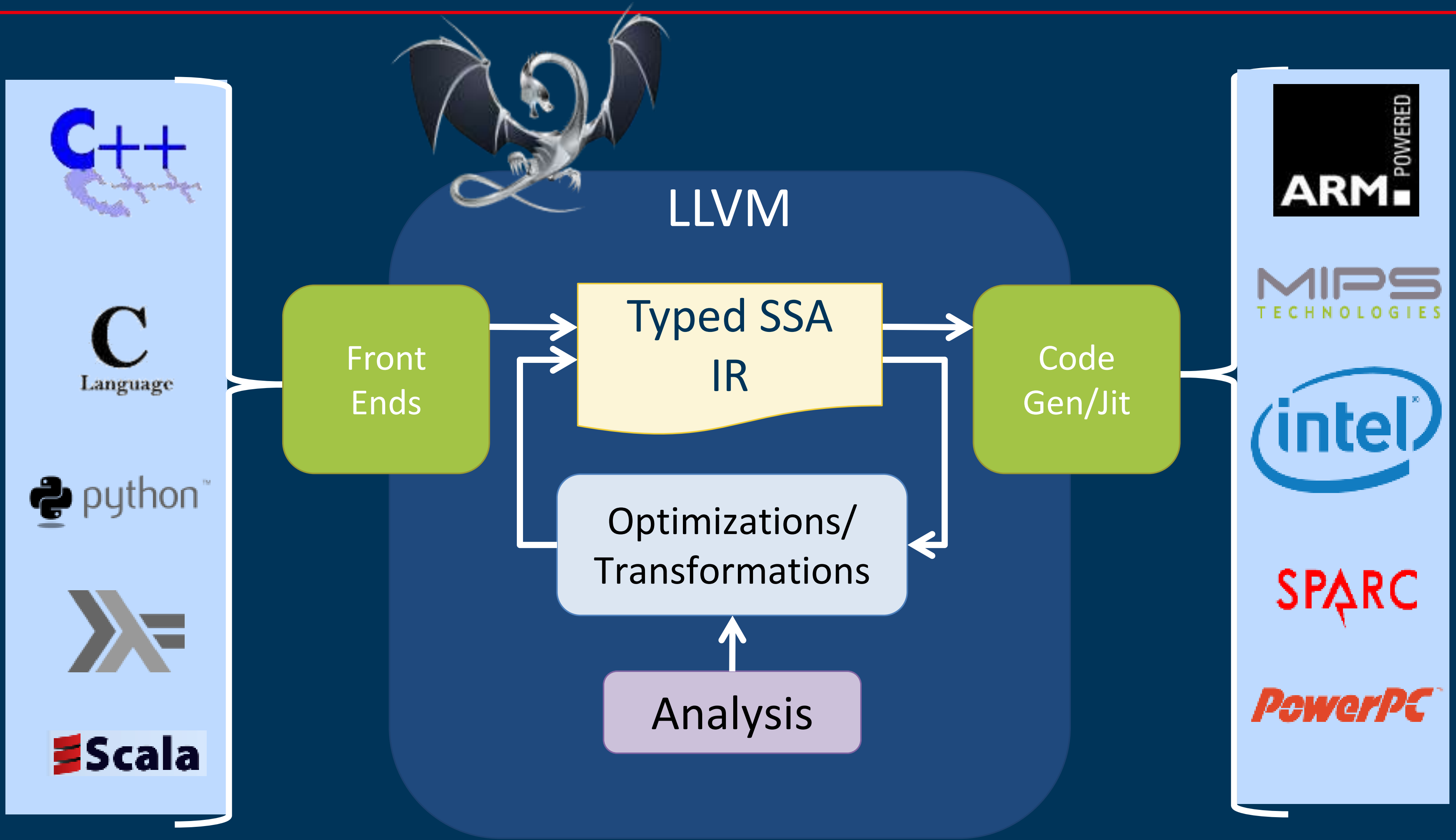


Diagram created by Steve Zdancewic, University of Pennsylvania

LLVM IR

clang -S -emit-llvm

```
int64_t fact(int64_t n) {
  int64_t a = 1;
  for (; n > 0; n--)
    a *= n;
  return a;
}
```

```
define i64 @fact(i64) #0 {
  %2 = alloca i64, align 8
  %3 = alloca i64, align 8
  store i64 %0, i64* %2, align 8
  store i64 1, i64* %3, align 8
  br label %4

; <label>:4: ; preds = %11, %1
  %5 = load i64, i64* %2, align 8
  %6 = icmp sgt i64 %5, 0
  br i1 %6, label %7, label %14

; <label>:7: ; preds = %4
  %8 = load i64, i64* %2, align 8
  %9 = load i64, i64* %3, align 8
  %10 = mul nsw i64 %9, %8
  store i64 %10, i64* %3, align 8
  br label %11
```

LLVM: Control-flow graphs

LLVM programs must be structured as SSA CFGs

- Statements: assignments to temporaries, stores, loads
- Terminators: branches, returns

- LLVM computes CFG structure
- Much of the block structure is *implicit* in textual IR
  - But we'll be explicit in our representation of LLVM

```
define i64 @fact(i64) #0 {
  %2 = alloca i64, align 8
  %3 = alloca i64, align 8
  store i64 %0, i64* %2, align 8
  store i64 1, i64* %3, align 8
  br label %4

; <label>:4:    ; preds = %11, %1
  %5 = load i64, i64* %2, align 8
  %6 = icmp sgt i64 %5, 0
  br i1 %6, label %7, label %14

; <label>:7:    ; preds = %4
  …
  br label %11

; <label>:11:   ; preds = %7
  …
  br label %4

; <label>:14:   ; preds = %4
  %15 = load i64, i64* %3, align 8
  ret i64 %15
}
```

LLVM: Storage

LLVM has four classes of storage

- *Local variables (temporaries)*
- *Abstract locations (stack-allocated)*
- *Global declarations*
- *Heap-allocated*

LLVM: Storage

Local variables (temporaries) `%uid`

- Defined by instructions `%uid = …`
- Abstract version of machine registers
- Values don't change during execution
- Must satisfy *single static assignment*
  - Each `%uid` appears to the left of exactly one assignment in the entire CFG
  - Can extend SSA to allow richer use of locals—using φ-nodes

LLVM: Storage

Abstract locations (stack-allocated)

- Abstract version of stack slots
- Created using `alloca` instruction
  - Returns a reference, stored in a temporary:

```
%ptr = alloca i64
```

  - Amount of space determined by type
- Accessed using load and store instructions

```
store i64 42, i64* %ptr
%z = load i64, i64* %ptr
```

How do you like type tags?

LLVM: Storage

- *Global declarations* `@gid`
  - Single declaration `@gid = …`
  - Used to store "constant" strings, arrays, &c.
  - Not necessarily constant!
- *Heap-allocated*
  - Handled entirely throught external library calls
  - Runtime-dependent: malloc-like for compiling C-like languages, GC-like for memory-managed languages

STRUCTURED DATA: STRUCTS

Structured data

C has (roughly) three forms of structured data

- Structs
- Arrays (big structs)
- Unions

Common questions: layout, access patterns

LLVM has roughly parallel constructs

- No unions… how do you like pointer casts?

Common access operator: `getelementptr`

Compiling structs

How to compile this code?

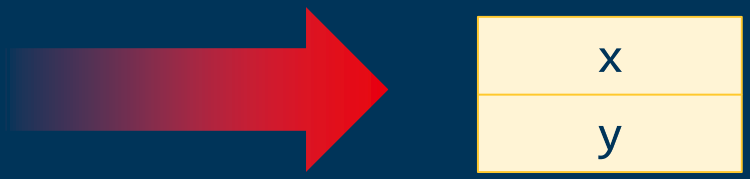
- How are points/rects represented in memory?
- How are accesses to structures compiled?
- How do we pass structures to functions?
- How do we return structures?

```
struct Point { int64_t x; int64_t y; };

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

struct Rect
mk_square(struct Point ll, int64_t 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

Basic idea: represent data contiguously in memory

```
struct Point {
  int64_t x, y;
}
```



Representing structs

Basic idea: represent data contiguously in memory

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

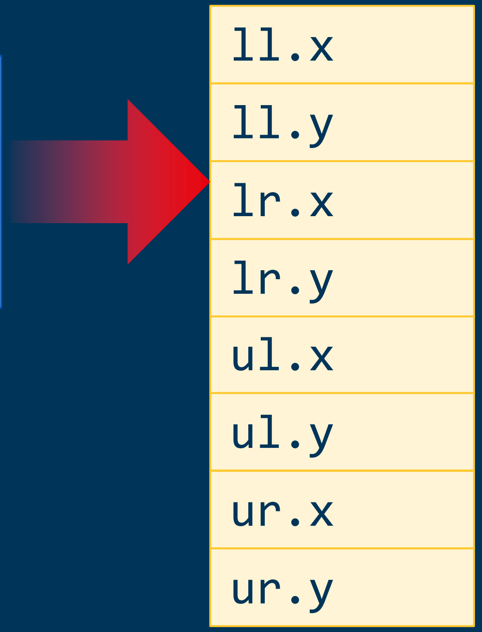
ll.x
ll.y
lr.x
lr.y
ul.x
ul.y
ur.x
ur.y

Accessing struct fields

Compiler has to know:

- Size of struct—to allocate
- Shape of struct—to access

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



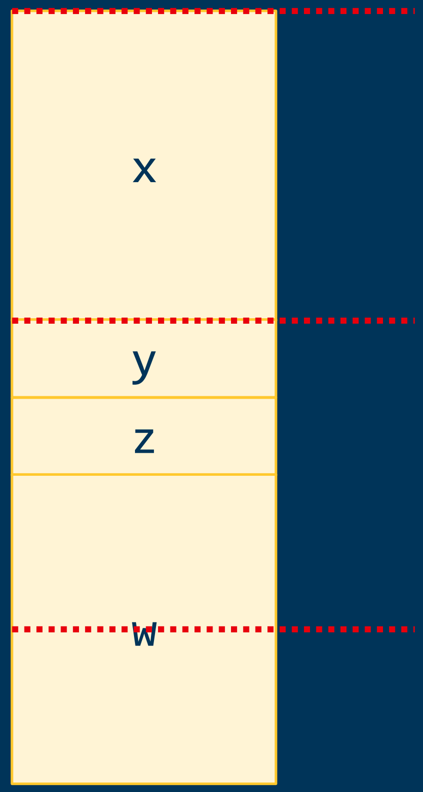
Can build nested access *by composition*

- `pt.x` = 0 offset, `pt.y` = 8 offset
- `rect.ll` = 0 offset, `rect.lr` = 16 offset, &c.
- `rect.lr.y` = 16 + 8 = 24 offset.

Representing structs: alignment

What if not all elements of a struct are the same size?

```
struct S {
  int64_t x;
  char y, z;
  int64_t w;
}
```

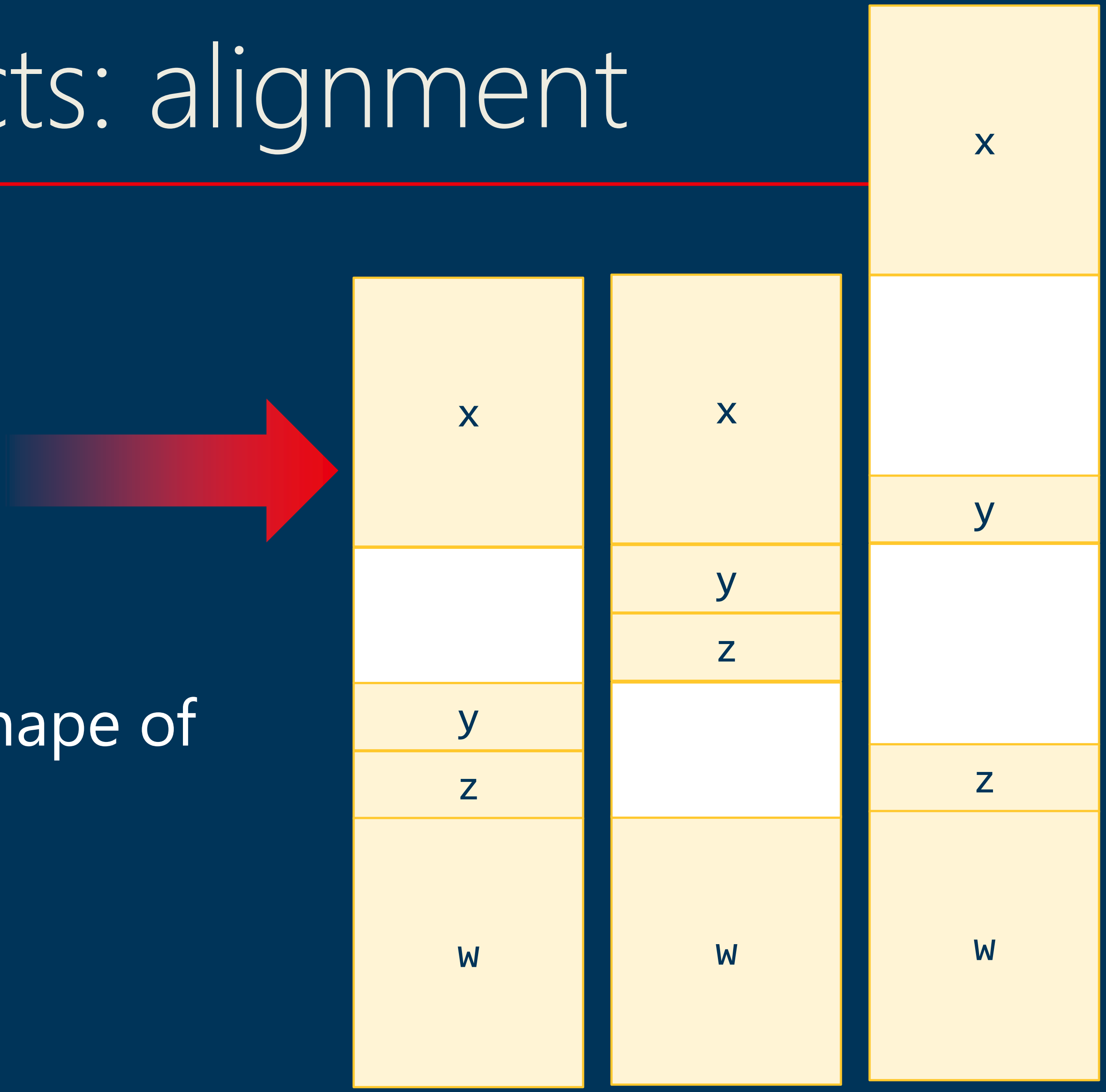


“Prefer” aligned data access

Representing structs: alignment

Approaches to packing fields:

```
struct S {
  int64_t x;
  char y, z;
  int64_t w;
}
```



- Has consequences for size/shape of structs
- Abstracted by LLVM

Structs: function arguments

What to do about struct arguments to functions?

- Split across multiple registers
- Copy struct into callee's memory

```
void printPoint(struct Point pt) {
  printf("%lld, %lld", pt.x, pt.y);
}
```

Copying structs: basically equivalent to series of assignments

- – May generate call to memcpy instead
- – Why "call-by-value" is bad terminology; "call-by-copying" instead?

Structs: function arguments

LLVM makes copying explicit.

```
void printPoint(struct Point pt) {
  printf("%lld, %lld", pt.x, pt.y);
}
```

```
define void @printPoint(i8*, %struct.Point*) #0 {
  …
  %4 = getelementptr %struct.Point, %struct.Point* %1, i32 0, i32 1
  %5 = load i64, i64* %4, align 8
  %6 = getelementptr %struct.Point, %struct.Point* %1, i32 0, i32 0
  %7 = load i64, i64* %6, align 8
  …
}
```

Structs: function returns

What to do about functions returning structs?

- Caller allocates space for result
- Callee copies struct into caller's stack space (possibly with memcpy)

```
struct Point makePoint(int64_t x, int64_t y) {
  struct Point pt = { x, y };
  return pt;
}
```

Again, "call-by-copying" instead of "call-by-value"

Structs: function returns

LLVM makes copying explicit:

```
struct Point makePoint(int64_t x, int64_t y) {
  struct Point pt = { x, y };
  return pt;
}
```

```
define void @makePoint(%struct.Point* noalias nocapture sret, i64, i64) #0 {
  %4 = getelementptr inbounds %struct.Point, %struct.Point* %0, i64 0, i32 0
  store i64 %1, i64* %4, align 8
  %5 = getelementptr inbounds %struct.Point, %struct.Point* %0, i64 0, i32 1
  store i64 %2, i64* %5, align 8
  ret void
}
```

Structs: pass-by-reference

Can avoid pass-by-value at the source level:

- Cost of passing struct is 1 word (equivalent to “real” cost)
- No copying
- Changes visible non-locally
- Return-by-reference more difficult

```
void printPoint(struct Point *pt) {
  printf("%lld, %lld", pt->x, pt->y);
}
```

```
void makePoint(struct Point *pt,
    int64_t x, int64_t y) {
  pt->x = x;
  pt->y = y;
}
```

ARRAYS (AKA BIG, UNIFORM STRUCTS)

One-dimensional arrays

- Stack-allocated (*used to* require knowing size at compile time)
- No alignment issues: all values same size.
- Indexing is "just" pointer addition

```
buf[i] = (buf + i * sizeof(*buf))
```

```
void foo() {
  int a[] = { 2, 6, 1, 0 };
  printf("%d\n", a[2]);
  printf("%d\n", *(a + 2));
  printf("%d\n", 2[a]);
}
```

Multi-dimensional arrays

Some languages support *multi-dimensional* arrays

- C: `int M[a][b]` gives an $a \times b$ length array, laid out by rows.
- Still "just" pointer addition (what is `M[i][j]`?)

M[0][0]
M[0][1]
M[0][2]
…
M[1][0]
M[1][1]
…
M[a-1][b-1]

```
void foo() {
  int a[][3] = { { 3, 6, 1 }, { 2, 8, 0 } };
  printf("%d\n", a[1][1]);
  printf("%d\n", *(a + 4));
  printf("%d\n", 1[a][1]);
  printf("%d\n", 1[1][a]);
}
```

Multi-dimensional arrays

Some languages support *multi-dimensional* arrays

- FORTRAN: `integer(a,b) :: m` gives an $a \times b$ length array, laid out by columns.
- Also: some C math libraries (inspired by FORTRAN libraries)

Why does row-major vs column-major order matter?

`m(0,0)`
`m(1,0)`
`m(2,0)`
…
`m(0,1)`
`m(1,1)`
…
`m(a,b)`

Array bounds

Safe languages check array bounds before accessing elements

- Need access to size of array
  - Common approach: store before first element
  - Pascal: only allow statically known array sizes
  - What about n-dimensional matrices?

`size`
`a[0]`
`a[1]`
…
`a[n-1]`

Array bounds

Safe languages check array bounds before accessing elements

- Compiler automatically inserts bounds checks before array accesses
- Decreases performance
  - Extra memory traffic
  - Extra jump
- Fertile ground for optimization

```
  movq -8(%rbx), %rdx
  cmpq %rdx, %rcx
  j l ok
  callq __explode
ok:
  movq (%rbx, %rcx, 0), %rax
```

STRUCTURED DATA IN LLVM

LLVM types

LLVM uses type tags (everywhere!) to capture the structure of data

τ	::=	`void`	
	|	`i1	i8	..	i64`	*n*-bit integers
	|	`[n x` τ`]`	Arrays
	|	$\tau(\tau_1, \tau_2, \ldots, \tau_n)$	Functions
	|	$\{\tau_1, \tau_2, \ldots, \tau_n\}$	Structures
	|	$\tau*$	Pointers
	|	`%T`	Named types

LLVM types

Types can be defined at the top level:

```
%struct.Point = type { i64, i64 }
```

- Named types can be recursive (via pointers)
- Actually just aliases to existing types

LLVM types

Example LLVM types

- `[42 x i64]`
- `[6 x [7 x i64]]`
- `{i64, [0 x i64]}`
- `%Node = {i64, %Node*}`

Computing pointers in LLVM

Pointer arithmetic (arrays and structs) abstracted by `getelementptr` instruction

- Given a pointer and series of indices, computes the indexed value
- Abstract equivalent of LEA—does *not* load the final (or any intermediate) pointer
- Multiple GEP's may be necessary to interpret a single C-style access

Computing pointers: examples

```
struct Point {
  int64_t x, y;
};
```

```
void printPoint(struct Point pt) {
  printf("%lld, %lld\n", pt.x, pt.y);
}
```

```
%struct.Point = type { i64, i64 }
```

```
define void @printPoint(%struct.Point*) {
  %3 = getelementptr %struct.Point, %struct.Point* %1, i64 0, i32 1
  %4 = load i64, i64* %3, align 8
  %5 = getelementptr %struct.Point, %struct.Point* %1, i64 0, i32 0
  %6 = load i64, i64* %5, align 8
  …
}
```

Computing pointers: example

```
%3 = getelementptr %struct.Point, %struct.Point* %1, i64 0, i32 1
```

- First argument: *type* of value being indexed
- Second argument: *pointer* to value being indexed (with type tag, for reasons)
- Remaining arguments: “path” into indexed value
  - First index: dereference pointer (think of it as `%1[0]`)
  - Struct indexes: must be i32, compile-time constant

Computing pointers: examples

```
int64_t a[] = { 3, 6, 1, 2, 8, 0 };
printf("%lld\n", a[3]);
```

```
%13 = getelementptr [6 x i64], [6 x i64]* %2, i64 0, i64 3
%14 = load i64, i64* %13
```

Computing pointers: example

```
int64_t indexer(int64_t a[][3], int b, int c) {
  return a[b][c];
}
```

```
define i64 @indexer([3 x i64]*, i64, i64) {
  %4 = getelementptr [3 x i64], [3 x i64]* %0, i64 %1, i64 %2
  %5 = load i64, i64* %4
  ret i64 %5
}
```

Computing pointers: example

```
%4 = getelementptr [3 x i64], [3 x i64]* %0, i64 %1, i64 %2
```

- Array indexing
  - Can be any integer type, determined at run-time
  - Sizes irrelevant (except for multi-dimensional arrays)
  - Convert freely between `[n x` τ`]` and τ*