# CSC488/2107 Winter 2019 — Compilers & Interpreters

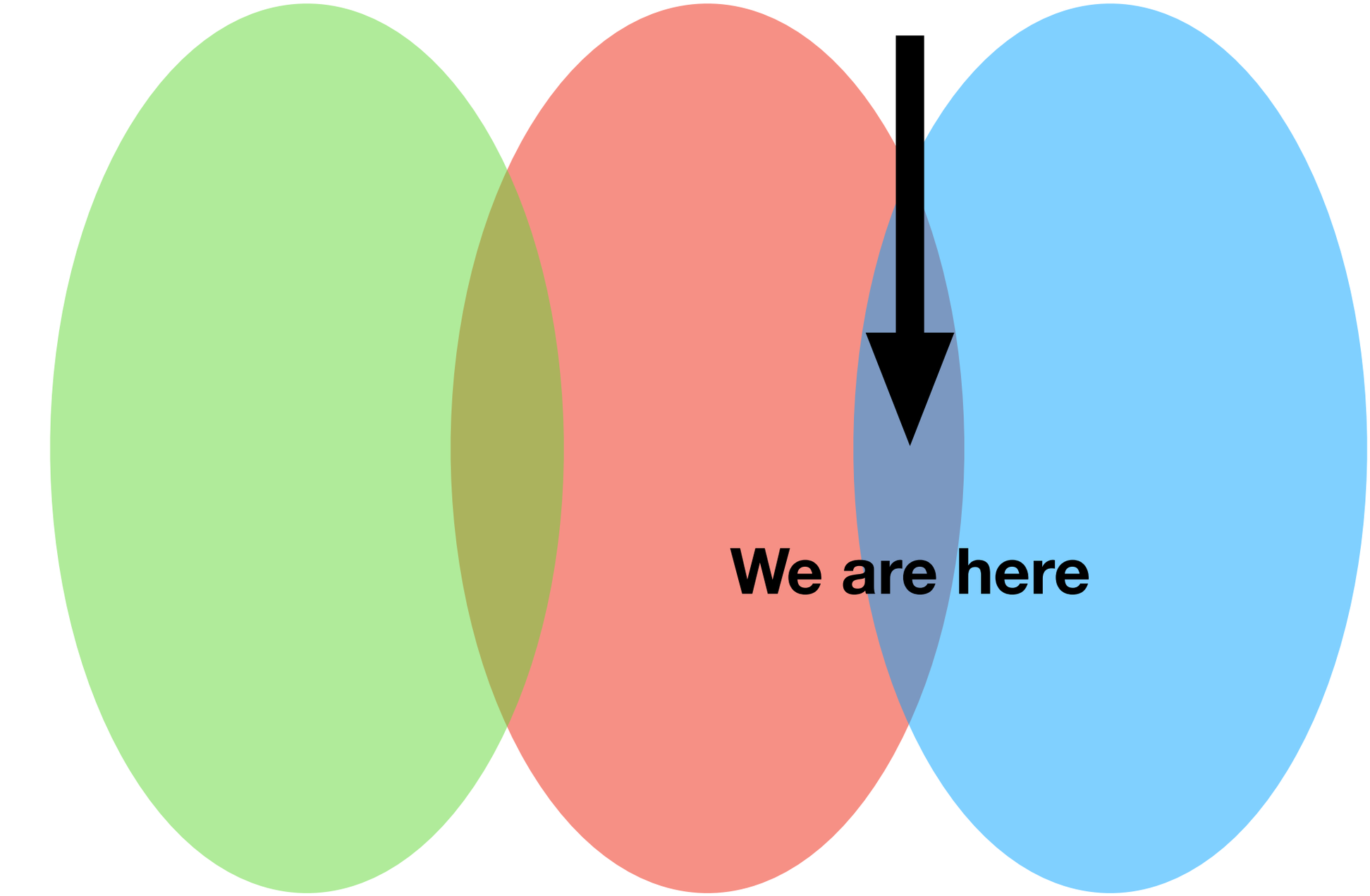
https://www.cs.toronto.edu/~csc488h/

Peter McCormick
pdm@cs.toronto.edu

# Agenda

- Runtime storage requirements
- Array storage
- CSC488 Machine

Recognize Analyze Transform




Frontend Backend

# Transform (Lower)

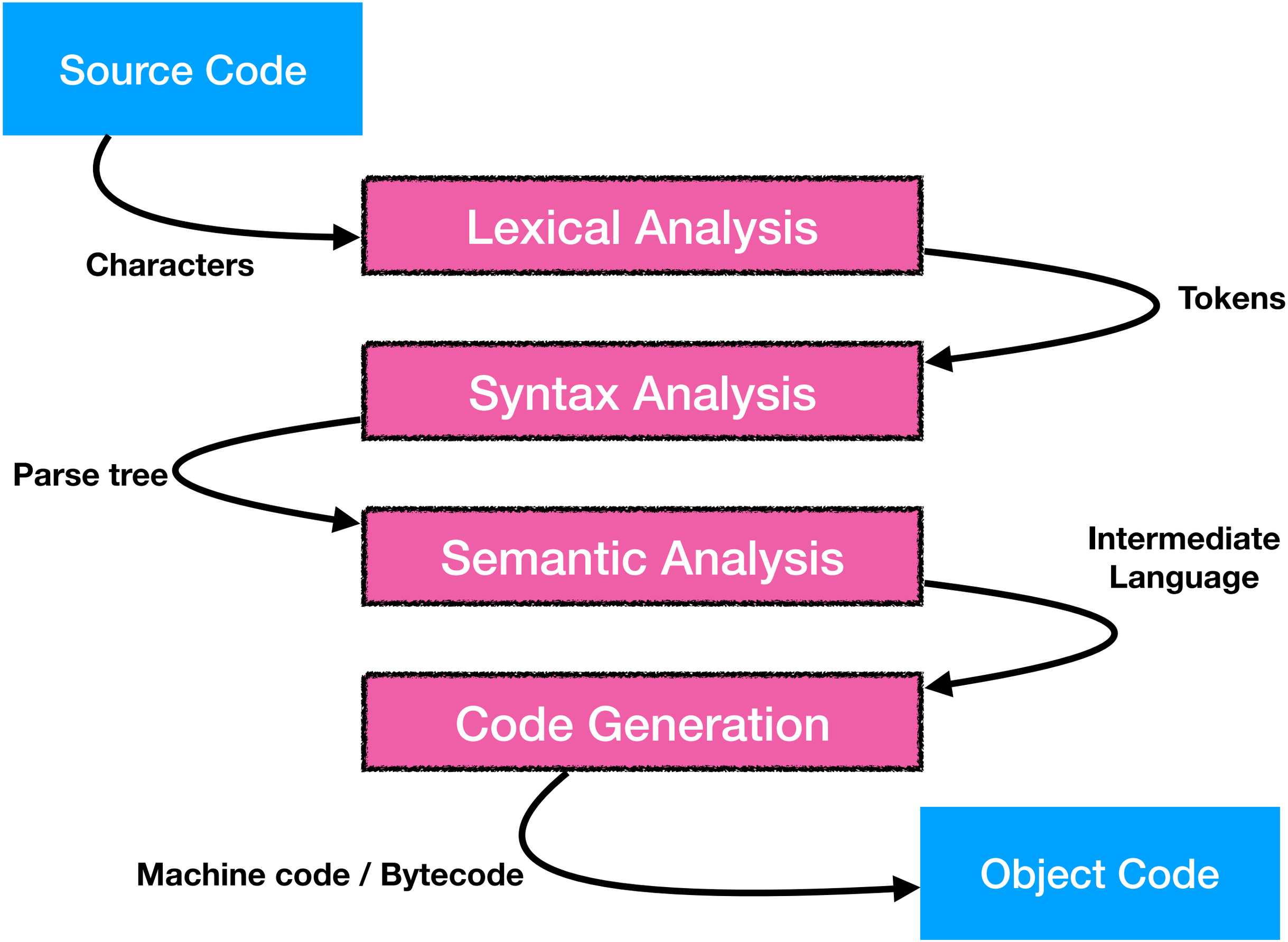
- Memory layout
- Optimization (optional)
- Code generation
- Very target specific

- Data flow:

  *Abstract Syntax Tree*

  ➥ *Intermediate Languages/Representations* (optional)

  ➥ *Target Machine Code*


Source Code
Characters
Lexical Analysis
Tokens
Syntax Analysis
Parse tree
Semantic Analysis
Intermediate Language
Code Generation
Machine code / Bytecode
Object Code

# Runtime storage requirements

- At runtime a program requires several kinds of storage:
  - Executable program *text* (machine code)
  - Global and static variables
  - Stack
  - Heap

# Program text

- The *linker* will collect all function object code from all *translation units*
    - Each *.c file is a *translation unit*, generating a *.o
- Linker will resolve all cross-references
- Machine code placed together in a memory section marked as *executable* (and typically read-only)
- Any unresolved references may be satisfied by shared libraries

# Global and static variables

- The *linker* will collect all global and static variables from all translation units

- Variables will be packed together in large contiguous block of memory

- Global constants will typically be placed into a separate read-only *constant pool*

# Global and static variables

```
// Global variable and constant
char *course = “CSC488”;


// Elsewhere…
course[0] = ‘.’;        // fault
                        // (constant RO)

course    = “…”;        // okay
```

# Global and static variables

```
/*
 * Global, but only visible
 * within this translation unit
 */

static int G = 488;
```

# Global and static variables

```
int incr()
{
    /*
     * Global, but only visible
     * within this function scope.
     *
     * Not thread safe.
     */
    static int count = 0;

    return count++;
}
```

# Stack

- Stack used for all local variables of functions and procedures
  - Typically starts at high memory address and grows *downwards* (towards 0)
- Grows and shrinks dynamically based on function calls and returns
- Laid out in *activation frames*

# Heap

- A dynamically sized region that programs can request an allocation of memory from (think *malloc*)

- Runtime works with operating system kernel to satisfy a given request for the running process

- A *memory allocator* may manage the specific layout of the heap, subdividing it into smaller regions based on allocation request sizes

- A program may choose to return unused memory to the heap via manual *deallocation* (think *free*), or a runtime system may automatically find and return unused memory via *garbage collection*

# Array storage

# 1D Arrays

```
typ A[10];
```

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| A[0] | A[1] | A[2] | A[3] | A[4] | A[5] | A[6] | A[7] | A[8] | A[9] |

```
ADDR(A[i]) = ADDR(A)    + (i * sizeof typ)
           = ADDR(A[0]) + (i * sizeof A[0])
```

# 2D Arrays: Row vs Column Major Order

```
typ M[3][3];
```

***M[row][column]***

**Row Major** →

**Column Major** ↓

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

# 2D Arrays: Row vs Column Major Order

```
typ M[3][3];
```

**Row Major Order**

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| M[0][**0**] | M[0][**1**] | M[0][**2**] | M[1][0] | M[1][2] | M[1][2] | M[2][0] | M[2][1] | M[2][2] |

**Column Major Order**

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| M[**0**][0] | M[**1**][0] | M[**2**][0] | M[0][1] | M[1][1] | M[2][1] | M[0][2] | M[1][2] | M[2][2] |

# 2D Arrays: Row vs Column Major Order

```
typ M[3][3];
```

```
ADDR(M[i][j])
    = ADDR(M) + (offs * sizeof typ)

offs = (i * stride0) + (j * stride1)
```

***Stride: how large a* step *in a given direction***

# 2D Arrays: Row vs Column Major Order

```
typ M[3][3];
```

```
offs = (i * stride0) + (j * stride1)
```

**Row Major Order:** *M[i][j]*

| **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** |
|---|---|---|---|---|---|---|---|---|
| M[0][**0**] | M[0][**1**] | M[0][**2**] | M[1][0] | M[1][2] | M[1][2] | M[2][0] | M[2][1] | M[2][2] |

*stride0=3, stride1=1*

# 2D Arrays: Row vs Column Major Order

```
typ M[3][3];
```

```
offs = (i * stride0) + (j * stride1)
```

**Column Major Order:** *M[i][j]*

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| M[**0**][0] | M[**1**][0] | M[**2**][0] | M[0][1] | M[1][1] | M[2][1] | M[0][2] | M[1][2] | M[2][2] |

*stride0=1, stride1=3*

# 2D Arrays: Row vs Column Major Order

```
typ M[m][n];
```

```
ADDR(M[i][j])
    = ADDR(M) + (offs * sizeof typ)

offs = (i * stride0) + (j * stride1)
```

**Row Major Order**

*stride0=n, stride1=1*

**Column Major Order**

*stride0=1, stride1=m*

# Multidimensional Arrays

```
typ M[d0][d1]…[d_n];
```

```
ADDR(M[i0][i1]…[in])
    = ADDR(M) + (offs * sizeof typ)

offs = i0*s0 + i1*s1 + … + i_n*s_n
```

**Row Major Order**

```
s0   = d1*d2*d3*…*d_n
s1   =    d2*d3*…*d_n
s_j  = d(j+1)*d(j+2)*…*d_n
s_n  = 1
```

# Multidimensional Arrays

```
typ M[d0][d1]…[d_n];
```

```
ADDR(M[i0][i1]…[in])
    = ADDR(M) + (offs * sizeof typ)

offs = i0*s0 + i1*s1 + … + i_n*s_n
```

**Column Major Order**

```
s0   = 1
s1   = d0
s2   = d0*d1
s_j  = d0*d1*…*d(j-1)
s_n  = d0*d1*…*d(n-1)
```

# Compile time constants

**Compute *multiplier* (memory offset) from *stride* (element index)**

```
ADDR(M[i0][i1]…[in])
    = ADDR(M) + (offs * sizeof typ)

offs = i0*s0 + i1*s1 + … + i_n*s_n
```

```
ADDR(M[i0][i1]…[in]) = ADDR(M) + offs

offs = i0*mult0 + i1*mult1 + … + in*mult_n

mult_j = sizeof typ * s_j
```

# Calculating addresses

*Compute ADDR(M[…]) using two registers*

```
R_a ← base

R_b ← i_0
R_b ← R_b * mult_0
R_a ← R_a + R_b

R_b ← i_1
…
```

# Array storage

- Most languages default to *row major ordering*, while *column major ordering* is often found in graphics and scientific computing
    - Row major: rightmost subscript of consecutive array elements varies most rapidly
    - Column major: leftmost subscript varies most rapidly
- Addressing an array element requires knowing the declared dimensions of the array
- While array elements are indexed sequentially, the *address* of consecutive elements will vary in multiples of the size of the underlying unit storage

# CSC488 Machine

# CSC488 Machine

- Stack-based bytecode virtual machine
    - No registers, all operations interact with the stack
- Fundamental data unit is a 32 bit signed integer *word*
- Memory contains of 8 *mebiwords* (mebi=$2^{20}$) or 256 mebibits (Mib) of storage
- von Neumann architecture: single unified address space for code, constants and stack
- Stack starts at top address ($2^{23}$-1), grows downwards towards 0

# CSC488 Machine

- Each machine *instruction* is a single word
- Special *MACHINE_TRUE* and *MACHINE_FALSE* integer constants
- Internal registers:
    - **PC**: current program counter (address of current instruction)
    - **MSP**: machine stack pointer (address where *next* push will write to, one less than address of last pushed value)
- PC will be incremented by 1 after each successful instruction execution
    - With the possible exception of the control flow instructions
- Machine also has a special array called a *display*, containing 256 machine words

# Stack

- Any ***push*** of a value to the stack performs the following:
  - Write the value to the memory address pointed to by MSP
  - Decrement MSP by 1 *(msp'=msp-1)*
- And ***pop*** of an value from the stack performs the following:
  - Increment MSP by 1 *(msp'=msp+1)*
  - Read the value from the memory address pointed to by MSP
- The ***top*** value of the stack is at memory address MSP+1

# Instruction Set

**General**

NOP
HALT

**Stack**

PUSH <*const*>
POP
POPN
DUP
DUPN
SWAP

**Arithmetic**

ADD
SUB
MUL
DIV
NEG

**Logical**

EQ
GT
OR

**I/O**

PRINTC
PRINTI
PRINTB
READI

**Control Flow**

BR
BF

**Memory**

LOAD
STORE
PUSHMSP

**Display**

SETD <*LL*>
ADDR <*LL*> <*Offs*>

# Instruction Set — General

- **NOP**: do nothing
- **HALT**: stop the machine from running

# Instruction Set — Stack

- **PUSH <*const*>**: push a *24 bit signed* constant onto the stack
    - Cannot directly push a full 32 bit constant, i.e. *push 2147483647* is not valid
- **POP**: pop and discard the top value from the stack
- **POPN**: repeatedly pop values from the stack
    - Pop a value N, then pop and discard N more times

# Instruction Set — Stack

- **DUP**: duplicate the top value on the stack
  - Pop the top value, and then push it back *twice*
- **DUPN**: repeatedly duplicate a value
  - Pop a value N, then pop a value V, then push V onto the stack N times
- **SWAP**: swap the top 2 values of the stack
  - Pop a value X, pop a value Y, then push X, followed by pushing Y

# Instruction Set — Arithmetic

- **ADD**: pop top 2 values, add them and push the result
    - Pop a value R, pop a value L, compute L + R and push the result
- **SUB**: pop top 2 values, subtract them and push the result
    - Pop a value R, pop a value L, compute L - R and push the result

# Instruction Set — Arithmetic

- **MUL**: pop top 2 values, multiply them and push the result
    - Pop a value R, pop a value L, compute L * R and push the result
- **DIV**: pop top 2 values, integer divide them and push the result
    - Pop a value R, pop a value L, compute L / R and push the result
- **NEG**: pop a value, negate it, push the result
    - Pop a value X, compute -X and push the result

# Instruction Set — Logical

- **EQ**: compare top 2 values for equality
    - Pop top 2, if equal push MACHINE_TRUE, else push MACHINE_FALSE
- **GT**: perform greater-than comparison on top 2 values
    - Pop value R, pop value L, compute L > R and push corresponding machine boolean constant

# Instruction Set — Logical

- **OR**: logical *or* of top 2 values
    - Pop top 2, push MACHINE_TRUE if at least one of them is equal to MACHINE_TRUE

# Instruction Set — Logical

- No instructions for AND, or NEQ, or LT, LTE, GTE…
  - How can you synthesize these using the others?

# Instruction Set — I/O

- **PRINTC**: pop from the stack and print as an ASCII character

- **PRINTI**: pop from the stack and print as a signed 32 bit integer

- **PRINTB**: pop from the stack, and print “t” if equal to MACHINE_TRUE, “f” is equal to MACHINE_FALSE, otherwise “?”

# Instruction Set — I/O

- **READI**: read an integer from the user and push to stack
  - Decimal: 488, 2107, 2019
  - Hexadecimal: **0x**1e8, **0x**83b, **0x**7e3
  - Binary: **0b**111101000, **0b**100000111011, …
  - Malformed input will push 0

# Instruction Set — Control Flow

- **BR**: unconditional branch (jump)
    - Pop an address from the stack, and set the PC to that address
- **BF**: branch if condition false
    - Pop an address, pop a value, and if value is MACHINE_FALSE, and set the PC to that address; otherwise do nothing

# Instruction Set — Control Flow

- No **BT** instruction
  - How could you synthesize this?
  - Do you actually need it?

# Instruction Set — Memory

- **LOAD**: read a value from memory
    - Pop an address from the stack, read a value from that memory address, and push the value to the stack
- **STORE**: write a value to memory
    - Pop a value, pop an address, and write that value to that memory address

# Instruction Set — Memory

- **PUSHMSP**: push the current value of MSP to the stack
    - After executing this instruction, the top value on the stack is a value that corresponds to the memory address of that same value on the stack

# Instruction Set — Display

- The *display* is a special array of 256 machine words
- **SETD <*LL*>**: pop a value from the stack, and save it to *Display[LL]*
  - *LL* stands for lexic level, and is an unsigned 8 bit value (0 <= LL <= 255)

# Instruction Set — Display

- **ADDR <*LL*> <*Offs*>**: compute *Display[LL]*+*Offs*, and push it to the stack
    - *Offs* is a *signed* 16 bit value
    - *ADDR <LL> 0* pushes the value of *Display[LL]*, the same valued that *SETD* popped