Regex Practice Question

- Which of the following regexes match the given strings?
  ○ Regex A: a[bc]+d(ab|aa)?
  ○ Regex B: aa*b+c?da?b
  ○ Regex C: aa?b[abd]+

- Fill the table with Yes/No

<table>
<thead>
<tr>
<th>String</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>abbdada</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>aabcbadb</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>abada</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>aabbcdb</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>aabbdab</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Regex Practice Question: Answers

- Which of the following regular expressions match the given strings?
  - Regex A: a(bc)+d(ab|aa)?
  - Regex B: aa*b+c?da?b
  - Regex C: aa?b[abd]+

- Fill the table with Yes/No

<table>
<thead>
<tr>
<th>String</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>abbdadaa</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>aabcbadb</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>abada</td>
<td>N</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>aabbcdb</td>
<td>N</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>aabbdab</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>
After we have an Abstract Syntax Tree (AST) of the code:

- Check that all the types are correct
  - Ex: cannot multiply two strings
- Check scope
  - Ex: does the variable I’m using actually exist yet?
- Create symbol table
  - A list of all the variables, functions, classes, etc. in the program and their types

Finally we convert the AST to machine code

- Convert loops and conditionals to jumps and branches
Object Files

- The generated code is put in an object file
  - text segment
    - the machine code itself
  - data segment
    - any constant values
  - defined external symbols:
    - symbols (functions, variables, classes) in our code that other object files can use
  - undefined external symbols
    - symbols we need from other code
  - local symbols for debugging
Linker

● What happens if we have multiple files?
  ○ How do we know where to go to call functions in other code?

● The linker puts multiple object files together into an executable

● How do we call the functions in other object files?
  ○ All object files assume the addresses in the file start at zero.
  ○ When object files are combined the linker puts them at different places in memory.
    ■ It has to modify all the absolute addresses in the object file to match.
    ■ This is called relocation.
    ■ Relative addresses (branches, some jumps) are unaffected.
Types of Artifacts

- **Static Library**
  - Simple bundle of object files
  - Added into executables/shared libraries during compilation

- **Dynamic Library (Shared Library)**
  - Also a bundle of object files
  - Loaded into executables at run time
    - Has to use **dynamic linking** to find addresses of symbols.
  - `.dll` on windows, `.so` on linux, `.dylib/.so` on mac

- **Executable.**
  - Still just a bundle of object files!
  - Has a defined entry point where the program starts (main function)
  - `.exe` on windows
Interpreted Code

- Some programming languages don’t use a compiler
  - Ex: python, php, perl, javascript

- Instead we run a program that reads the source code file and:
  - Follows the instructions in it right away
  - Does not create any executable file
  - Same as a compiler, but instead of generating code, it runs the code right away

- Some programming languages do both!
  - Compile to an intermediate quasi-machine language (usually called bytecode)
  - An interpreter reads the bytecode and does the instructions
  - Ex: java, C#
Execution of Code

- When you run a program you start a process.
- Processes are the basic building blocks of the operating system.
- They are comprised of threads and an address space.
Threads

- A thread represents code executing somewhere in a process
  - Registers, program counter, etc.

- At any given time, exactly one thread is running on a CPU core
  - If you have a multi-core CPU (you most certainly do), then as many threads as there are CPUs can be running.

- A thread can be: **running**, **ready**, or **blocked**
  - **running** threads are currently executing code
  - **ready** threads are not running right now, but are able to be run immediately
  - **blocked** threads cannot be run because they are waiting for something else to happen
    - This is usually I/O of some kind (network, keyboard, mouse, touchscreen, etc.)
Thread Scheduling

- Threads that are runnable (in the **running** or **ready** states) take turns using a CPU core (algorithm decides who’s turn it is)
- When a thread has to wait, it **blocks** and the next thread starts its turn
- If a thread runs for some time (a **time slice**), it gets **preempted**
  - A preempted thread moves from the running state to the ready state.
Address Spaces

● Each process has its own address space
  ○ All memory addresses in a process are specific to that process
  ○ There can be different things at the same address in different processes
  ○ Each process has the entire range from 0 to \(2^{32}\) to itself (or \(2^{64}\), etc.)

● All threads in a process share this address space!
  ○ So you have to be careful not to overwrite memory other threads are using
  ○ This can lead to *lots* of bugs
Thread Operations

- Thread scheduling is non-deterministic
  - From the point of view of the programmer
  - When multiple threads are involved, things happen in weird orders

- Basic thread operations:
  - create
    - Starts a new thread in the current process at the given function address
    - Can pass a parameter to the function
  - join
    - Waits (blocks) until the specified other thread ends
Thread Examples 1

- Give all the possible outputs of the following code:

```c
main() {
    create(funcA, "q");
    create(funcA, "r");
    create(funcB, "s");
}

funcA(letter) {
    print(letter);
}

funcB(letter) {
    print(letter);
    print(letter);
}
```
Thread Examples 1: Solution

- Give all the possible outputs of the following code:

```c
main() {
    create(funcA, "q");
    create(funcA, "r");
    create(funcB, "s");
}

funcA(letter) {
    print(letter);
}

funcB(letter) {
    print(letter);
    print(letter);
}
```

- qrss, qsrs, qssr, sqrs, sqsr, srsq, srqs, ssrq, ssqr, rsqs, rssq, rqss
  - It’s all the combinations! We have no ordering here.
Thread Examples 2

- Give all the possible outputs of the following code:

```c
main() {
    create(funcA, "q");
    create(funcA, "r");
}

funcA(letter) {
    print(letter);
    t1 = create(funcB, "w");
    join(t1);
    print(letter);
}

funcB(letter) {
    print(letter);
}
```
Thread Examples 2: Solution

- Give all the possible outputs of the following code:

```
main() {
  create(funcA, "q");
  create(funcA, "r");
}
funcA(letter) {
  print(letter);
  t1 = create(funcB, "w");
  join(t1);
  print(letter);
}
funcB(letter) {
  print(letter);
}
```

- `qwqrwr, rwrqwq, qrwwqr, rqwwrq, qwrqwr, rwqrwq, qwrwrq, rwqwqr, qrwrwq, rqwqwq, rqwwwr, qrwwwr`  
  - All the combinations where `qwq` and `rwr` are interleaved but still in order.
  - The only order we know for sure is `qwq` and `rwr`, because the `funcA` thread waits for the `funcB` thread to end before printing the letter again.
Assignment reminders

● Submit a .txt XOR a .pdf for each question
  ○ Do not submit both for the same question!
  ○ You may submit a .pdf for one question and a .txt for a different question

● Make sure your diagrams and tables are clear and easy to read
  ○ Make sure to leave enough space