Structure of a Compiler

Front End

- Source Language
- Lexical Analyzer
- Syntax Analyzer
- Semantic Analyzer
- Int. Code Generator

Intermediate Code

Back End

- Code Optimizer
- Target Code Generator
- Target Language

Projects:
- Project 1
- Project 2
- Project 4 & 5
- Project 3

Today!
No variables in TubeCode - you must manage your own data with a memory space and registers.

**Removed Instructions**: All array (ar_*) instructions.

**Removed Arguments**: scalars and arrays.

**Added Instructions**: load, store, and mem_copy

**Added Arguments**: 8 registers (regA through regH), memory locations

**Your job**: Automate the conversion of your programs!
New Tubecode Assembly Commands

- **load**  
  arg1  arg2  
  loads a value from the address \texttt{arg1} into register \texttt{arg2}

- **store**  
  arg1  arg2  
  stores the value of \texttt{arg1} into the address denoted by \texttt{arg2}

- **mem_copy**  
  arg1  arg2  
  copies the value stored in memory at address \texttt{arg1} to the address \texttt{arg2}
What is the value value in memory position 1?

1.5
2.4
3.1
4. Unknown

val_copy 4 regA 1.5
store regA 5 2.4
load 5 regA 3.1
store regA 1 4. Unknown
What is the output of the TCA program?

store 15 4
val_copy 4 regA
load regA regB
load regB regC
out_val regC

1. 15
2. 4
3. 1
4. Unknown
You must convert your Tube Intermediate Code to TubeCode Assembly. The intermediate code instructions should be stored internally, but you may read them back in from a temporary file if it’s automatic.

You have only 8 registers to work with. Each variable must have a dedicated memory position for when it cannot be in a register.

**Before each instruction**: load input-variables into registers.

**After each instruction**: store output-variables in memory.
Example Conversion

**Tubular:**
val x = random(10) - 5
print(x);

**AST:**
```
program
  =
    s1
      =
        x
        s5
          -
            s4
              5
              random
                s2
                  10
```

**Tube Intermediate Code:**
```
val_copy 10 s2
random s2 s3
val_copy 5 s4
sub s3 s4 s5
val_copy s5 s1
out_val s1
out_char '\n'
```

*Done! (with IC)*
Tubular:

```javascript
val x = random(10) - 5
print(x);
```

**AST:**

```
program
  =
  print
    x
    -
    x
    random
      10
    5
```

**Tube Intermediate Code:**

```python
val_copy 10 s2
random s2 s3
val_copy 5 s4
sub s3 s4 s5
val_copy s5 s1
out_val s1
out_char '\n'
```

**TubeCode Executable:**

```python
val_copy 10 regB
store regB 2
load 2 regA
random regA regB
store regB 3
val_copy 5 regB
store regB 4
load 3 regA
load 4 regB
sub regA regB regC
store regC 5
load 5 regA
val_copy regA regB
store regB 1
load 1 regA
out_val regA
out_char '\n'
```
Track argument types for each intermediate code instruction as you translate it.

**Number literals, character literals, and labels** - These behave the same in TubeCode and can be moved directly.

**Scalar variables** - Must be replaced by registers and loaded from or saved to memory when used.

**Array variables** - These can only be used in well-defined places, but can be complicated to convert.
Create a collection of instruction definitions. Indicate read arguments, write arguments, and any other special notes about each instruction.

<table>
<thead>
<tr>
<th>Name</th>
<th>arg1</th>
<th>arg2</th>
<th>arg3</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>read</td>
<td>read</td>
<td>write</td>
<td></td>
</tr>
<tr>
<td>val_copy</td>
<td>read</td>
<td>write</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jump</td>
<td>read</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ar_get_idx</td>
<td>read</td>
<td>read</td>
<td>write</td>
<td>Expand to multiple instructions</td>
</tr>
<tr>
<td>test_less</td>
<td>read</td>
<td>read</td>
<td>write</td>
<td></td>
</tr>
</tbody>
</table>
Which arguments are read from or written to?

- random arg1 arg2
- jump_if_n0 arg1 arg2
When we compile arrays into TubeCode, we need to be able to store its size and its contents in memory.

For example, if an array has 0 through 9 in it:

```
10 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

This is easy if we know everything about an array at compile time, but what about variable sized arrays? And resizing? How do we handle these?
Managing TubeCode Arrays

We can handle all memory allocation for scalar variables at compile time, but not so for arrays.

If the size of an array changes, we may need to change the location of the array in memory.

We can keep indirect references to arrays at a fixed memory position that points to the start of the array. If the array moves, we update this reference.
Since arrays can change dynamically, we must implement run-time memory management.

A crude-but-easy solution: dedicate a fixed memory position to point to the next unused memory slot. When we need new memory, start from that recorded position and then update the free-memory pointer.

This is very inefficient, but correct!
Converting Array Instructions

Five intermediate code instructions are not available in TubeCode:

- `ar_get_idx`
- `ar_set_idx`
- `ar_get_size`
- `ar_set_size`
- `ar_copy`

How should we convert each of these?
Converting Array Instructions

How to convert:

\texttt{ar\_get\_size a12 s35}

For each variable, we will assume that we keep its info at the associated memory position

\texttt{load 12 regA} \quad \# \text{ Load array pointer into regA}
\texttt{mem\_copy regA 35} \quad \# \text{ Copy size into destination}

This conversion is very simple... the first item in the array is its size.
Converting Array Instructions

How to convert:
```
ar_get_idx a12 s24 s35
```

```
load 12 regA  # Load array pointer into regA
load 24 regB  # Load index into regB
add regA 1 regA  # Increment regA past size
add regA regB regA  # Find position of index
mem_copy regA 35  # Copy value at index to destination
```
Converting Array Instructions

How to convert:

```
ar_set_idx a12 s24 s35
```

```
load 12 regA
# Load array pointer into regA
load 24 regB
# Load index into regB
add regA 1 regA  # Increment regA past size
add regA regB regA  # Find position of index
mem_copy 35 regA  # Copy value into the index location
```

Note: this is the same as `ar_get_idx`, except for the final line.
Converting Array Instructions

How to convert:

```
ar_set_size a12 s24
```

This conversion is, unfortunately, complex. What if the new size is larger than the current size?

If the size is the same or smaller, just change the recorded size at the beginning. If it is larger, however, we must move the array.

We can also keep it simple and *always* move the array.
ar_set_size a12 s24

load 12 regA  # Load pointer to array into regA
load 24 regB  # Load new array size into regB
load regA regC  # Load old array size into regC
store regB regA  # Replace old size and then test if done
store test_lte regB regC regD  # If new size < old size...
jump_if_n0 regD resize_end_1  # ...we are done! Jump to end
load 0 regD  # Load free mem position into regD
add regD 1 regE  # Set regE to first pos. in new array
add regE regB regE  # Move regE to new free mem position
store regE 0  # Store new free memory at pos. zero
store regD 12  # Redirect array pointer to new array pos.
store regB regD  # Store new size at new array start

resize_start_0:
add regA 1 regA  # Increment pointer to FROM array
add regD 1 regD  # Increment pointer to TO array
test_gtr regD regE regF  # If we are done copying...
jump_if_n0 regF resize_end_1  # ...jump to end of copy loop
mem_copy regA regD  # Copy the current index.
jump resize_start_0  # Jump back to start of copy loop.

resize_end_1:
Error Handling

When dealing with arrays, there are two run-time errors that may occur:

An **index out-of-bounds** error should occur when a Tubular program tries to index into an array before its beginning or after its end.

An **invalid memory request** error should be triggered if a `resize()` command tries to allocate negative memory.

**Extra Credit:** Implement these in a `debug mode`. 