Lecture 29:

- IR code generation (wrap up)
- Programming language paradigms

Announcements:

- HW-6 out
- Quiz 7 Fri: VM (trace instructions), code generation
(10) General rvalue path and lvalue expressions

- load the variable value (e.g., the \texttt{p} in \texttt{p.x.y.z})
- repeatedly add a \texttt{GETF} instruction for remaining path (e.g., \texttt{x}, \texttt{y}, and \texttt{z})
- for array access, generate index code and use \texttt{GETI}
- for assignment statements, last instruction is a \texttt{SETF} or \texttt{SETI}

(a) Simple example of non-array rvalue

```c
struct Node {
    int val;
    Node next;
}

void main() {
    Node p = new Node(3, null);
    int x = p.val;
}
```

Frame main

```
0: ALLOCS() // new Node
1: DUP()
2: PUSH(3)
3: SETF(val)
4: DUP()
5: PUSH(None)
6: SETF(next)
7: STORE(0) // p
8: LOAD(0)
9: GETF(val)
10: STORE(1) // x
11: PUSH(None)
12: RET()
```
(b) Simple example of lvalue ...

```c
struct Node {
    int val;
    Node next;
}

void main() {
    Node p = new Node(3, null);
    // circular linked list!
    p.next = new Node(0, p);
    p.next.val = 4;
}
```

Frame main

```
0: ALLOCS() // new Node
1: DUP()
2: PUSH(3)
3: SETF(val)
4: DUP()
5: PUSH(None)
6: SETF(next)
7: STORE(0) // p
8: LOAD(0)
9: ALLOCS() // new Node
10: DUP()
11: PUSH(0)
12: SETF(val)
13: DUP()
14: LOAD(0)
15: SETF(next)
16: SETF(next)
17: LOAD(0)
18: GETF(next)
19: PUSH(4)
20: SETF(val)
21: PUSH(None)
22: RET()
```

Left as an exercise:

- expressions (evaluated left to right; except for >= and >)
- if statements (similar to loops, but more jumps to keep track of)
- array access (similar to field access, but use GETI, SETI, gen index code)
**Programming Language Paradigms**

**Imperative vs Declarative Languages**

**Imperative Languages**: Programmers specify how to solve the problem and the system carries out the steps.

**Declarative Languages**: Programmers specify what the solution should look like and the system determines how best to compute the solution.

Logic and Functional languages are generally considered (more) declarative

- compared to object-oriented & procedural languages (C/C++/Python/Java/etc.)
- largely has to do with the underlying models of computation used

There are other ways languages are categorized as well

- e.g., script-based, object-oriented, dynamic vs static typing, memory-safety
From Turing Machines to Imperative Programming

Turing Machines:

1. **infinite tape** divided into memory cells (one symbol per cell)
2. **read/write head** that can move left/right one cell at a time
3. **state register** that stores the current state of the machine
4. **state transition table**:
   \[ \text{curr state } + \text{curr head symbol } \rightarrow \text{write symbol } + \text{new state } + \text{move head} \]

Example: replace a's with b's

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>b</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

↑

\[ s_i \]

the infinite tape

the tape head (with the machine in state \( s_i \))

Transition Table: (where \( s_1 \) is start symbol, \( s_2 \) is halt symbol)

<table>
<thead>
<tr>
<th>Current State</th>
<th>Current Symbol</th>
<th>New Symbol</th>
<th>New State</th>
<th>Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s_1 )</td>
<td>( a )</td>
<td>( b )</td>
<td>( s_1 )</td>
<td>Right</td>
</tr>
<tr>
<td>( s_1 )</td>
<td>( b )</td>
<td>( b )</td>
<td>( s_1 )</td>
<td>Right</td>
</tr>
<tr>
<td>( s_1 )</td>
<td>Blank</td>
<td>Blank</td>
<td>( s_2 )</td>
<td>Left</td>
</tr>
</tbody>
</table>

Turing Machines are imperative ...

- they specify how the computation should be carried out (very low level)
- inspiration for RAM machines (read from mem, do op, write to mem)
- where higher-level languages abstract from the low-level computation model