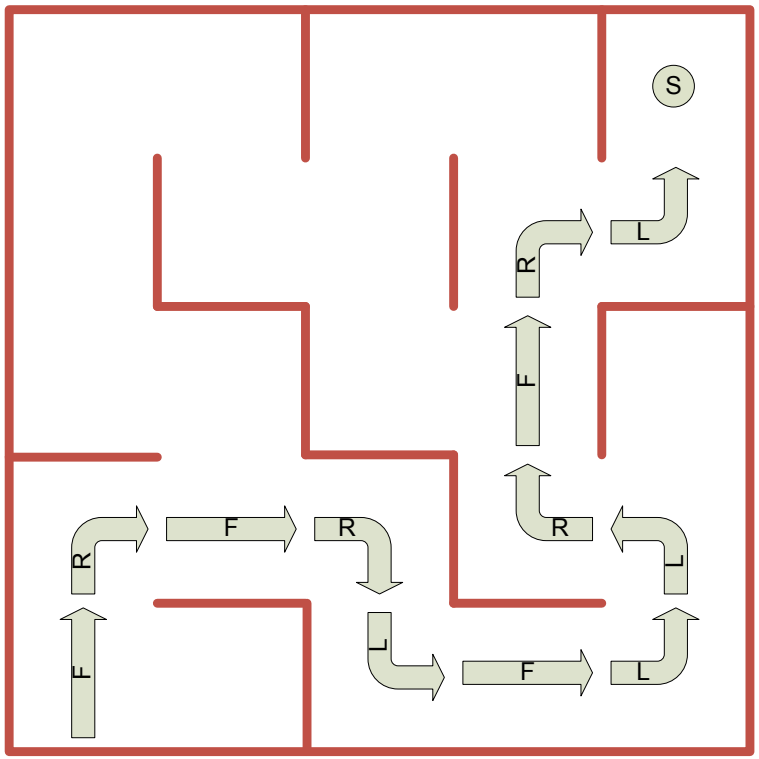
The Butterfly Machine

Diagonal Pathfinding
in a
maze of little twisty passages

Peter Harrison 2014

A Maze Path Example

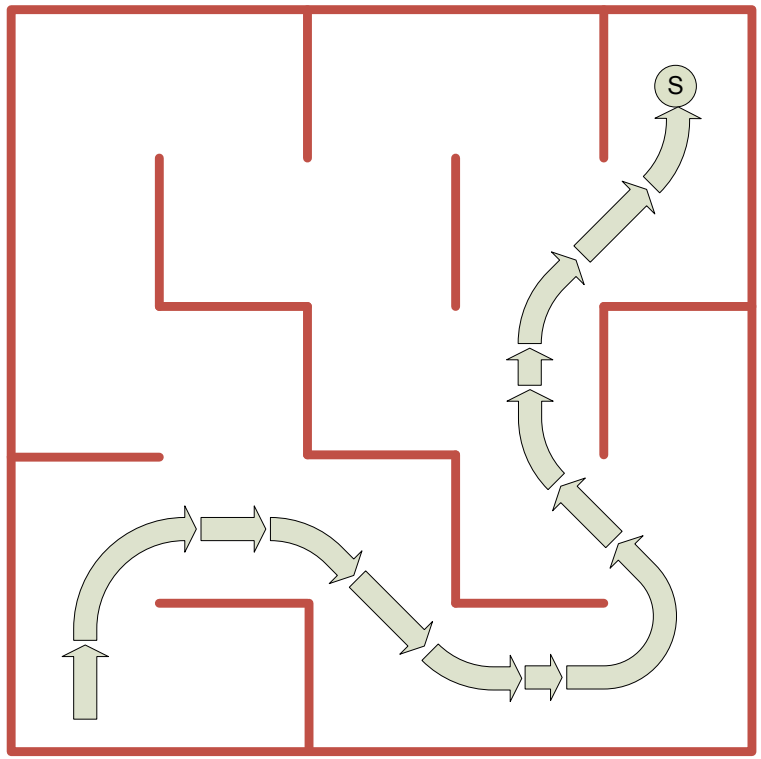
FRFRLFLLRFRLS



In-place turns, no run length encoding

A Maze Path Example

FRFLFLLRFRLS



Smooth turns, concatenated straights

```
FWD01
SS90SR
FWD02
SD45R
DIA02
DS45L
FWD02
SD135L
DIA02
DS45R
FWD02
SD45R
DIA02
DS45L
FWD01
STOP
```

Substring Searches

- Orthogonal path is a string
  - "FFRFFFLRFRLLFS"
- Run length encode
- Look for patterns in priority order

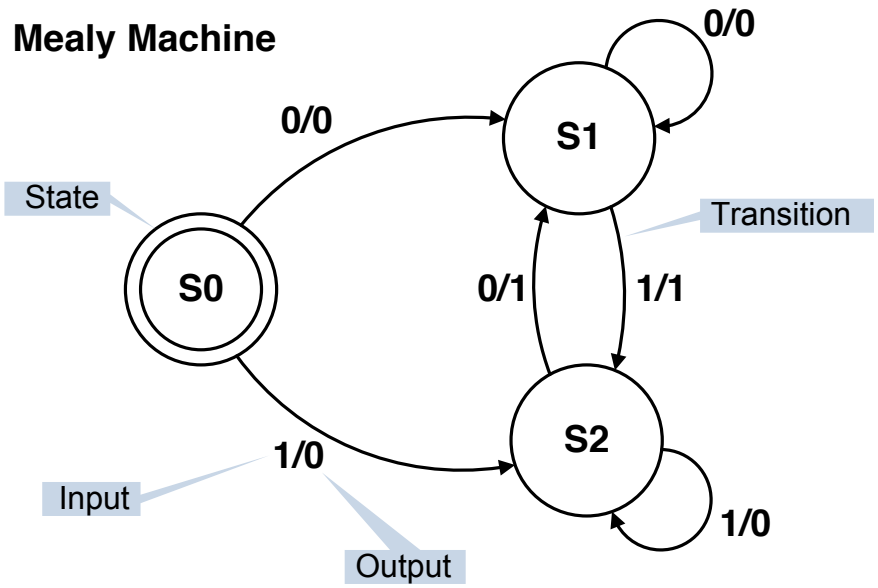
```
c0 = commandList[cptr + 0];
c1 = commandList[cptr + 1];
c2 = commandList[cptr + 2];
c3 = commandList[cptr + 3];
c4 = commandList[cptr + 4];
c5 = commandList[cptr + 5];
c6 = commandList[cptr + 6];
if ( (c1 > FWD1) && (c1 < FWD16)) {
  tempList[tptr++] = c1;
  cptr = cptr + 1;
} else if ( (c0 < FWD16) && (c1 == 'R') && (c2 == FWD1) && (c3 == 'L')) {
  tempList[tptr++] = SD45R;
  x = DIA1;
  cptr = cptr + 1;
} else . . .
```

Scan and Look Ahead

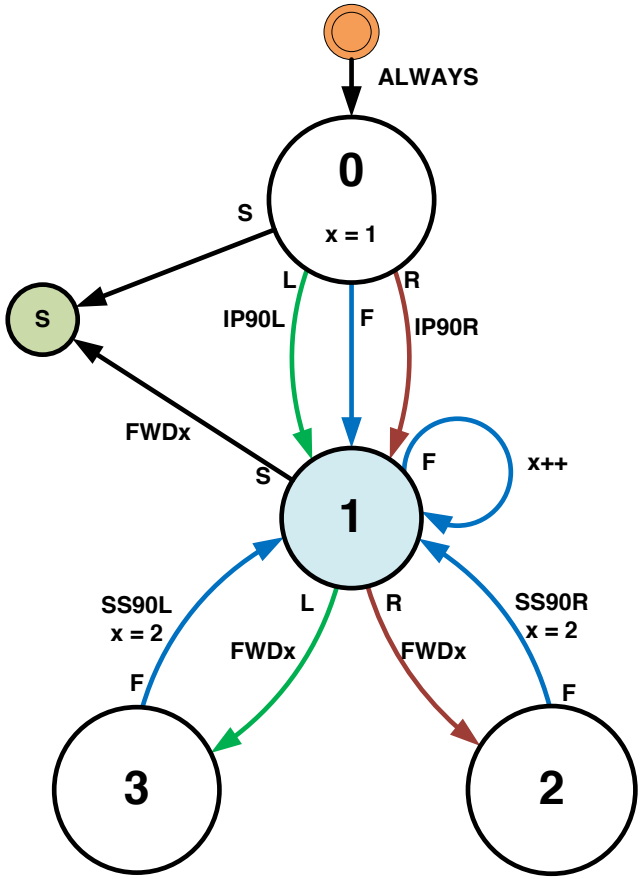
- Translate characters as found
- Look ahead as needed
- Retain straight vs diagonal state
- A single 'F' is unambiguous
- Turns need look ahead to translate

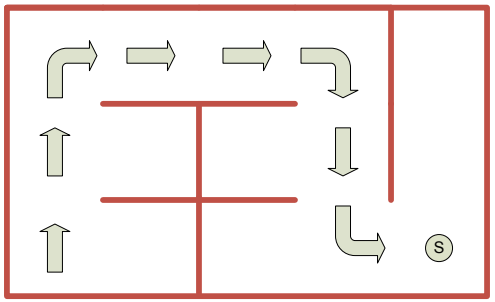
Translate as Found

- Effect of commands depends on history
- Classic case for a state machine
- Example



Orthogonal Machine






```
FFRFFRFLS

FWD02
SS90R
FWD03
SS90R
FWD02
SS90L
FWD01
STOP
```

Orthogonal Machine Implementation

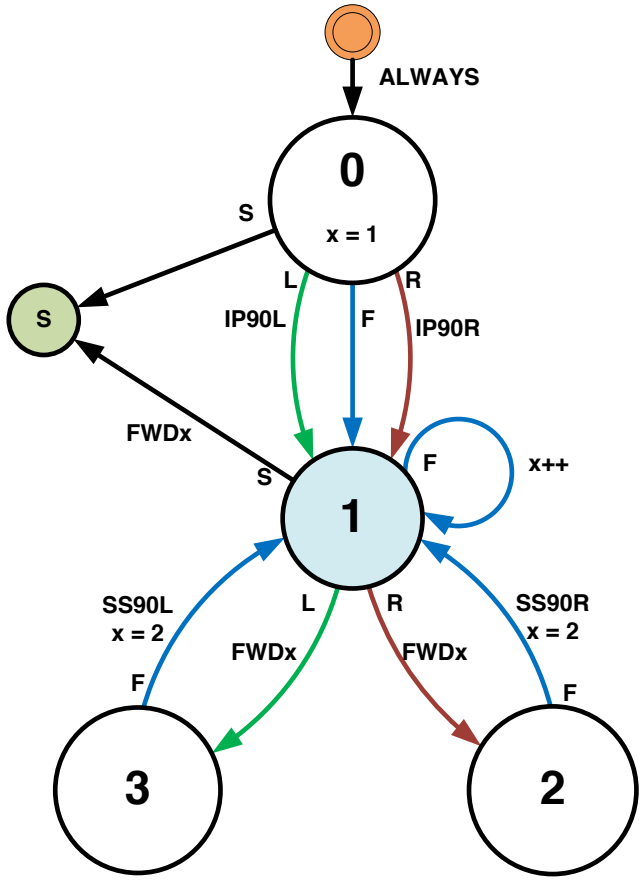
- Simple but lengthy switch/case statements

```
void orthoMachine(char * s) {
  int state = 0;
  unsigned char c;
  while (state != 11) {
    c = *s++;                    // fetch next instruction
    switch (state) {
      case 0:                    // entry point
        break;
      case 1:                    //F+
        break;
      case 2:                    //F+R
        break;
      case 3:                    //F+L
        break;
      case 10:                   // exit point
        emit(CMD_STOP);
        state = 11;
        break;
      default:                   // catch-all error state
        emit(CMD_ERROR);
        state = 11;
        break;
    }
  }
}
```

Orthogonal Machine Implementation

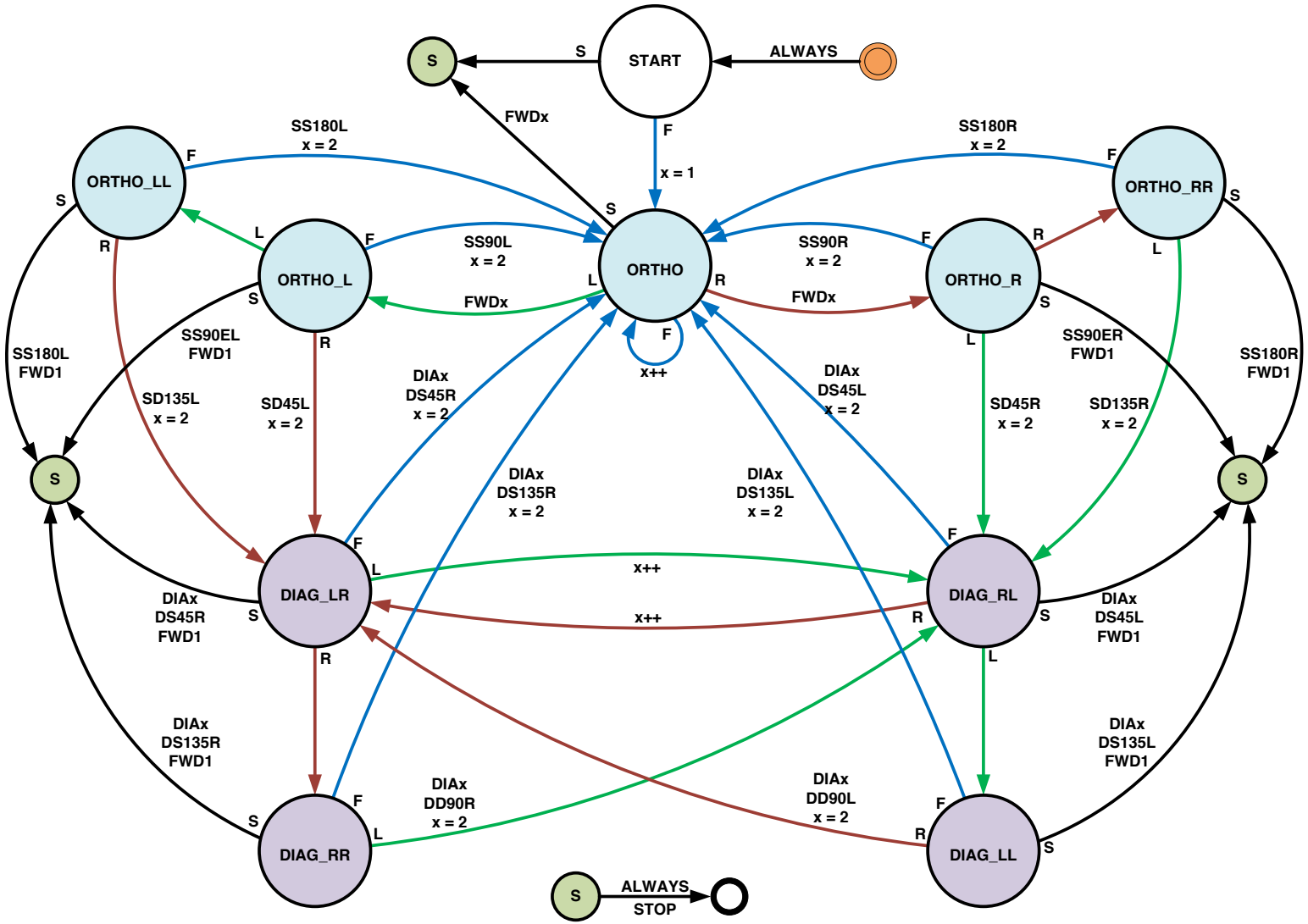
- State 1 code

```
case 1:
  if (c == 'F') {
    x++;
  } else if (c == 'L') {
    emit(FWD0 + x);
    state = 3;
  } else if (c == 'R') {
    emit(FWD0 + x);
    state = 2;
  } else if (c == 'S') {
    emit(FWD0 + x);
    state = 10;
  } else {
    state = 11;
  }
  break;
```



MAZE PATHFINDER STATE MACHINE

CONVERT ORTHOGONAL PATH TO DIAGONALS



Test Driven Development

- Problems:
  - State machine is complex
  - Short sequences not too hard
  - Real mazes forbidding with many paths
- Solutions:
  - Write a generic test function
  - Write simple tests from existing mouse code
  - Write code to make each test pass
  - Only progress when tests pass

Test Data

- Start simple, add complexity gradually
- Test data from existing mouse code (42 test pairs so far)

```
typedef struct {
  char input[MAX_LEN];
  COMMAND output[MAX_LEN];
} testPair_t;

testPair_t testPairs[] = {
  // basic straights
  {"S",       {STOP}},
  {"FS",      {FWD1, STOP}},
  {"FFS",     {FWD2, STOP}},
  // simple 90 degree turns
  {"FRS",     {FWD1, SS90ER, FWD1, STOP}},
  {"FLS",     {FWD1, SS90EL, FWD1, STOP}},
  {"FRFS",    {FWD1, SS90SR, FWD2, STOP}},
  {"FLFS",    {FWD1, SS90SL, FWD2, STOP}},
  {"FFFRFS",  {FWD3, SS90SR, FWD2, STOP}},
}
```

Test Process

- Test Function
  - Gets input string and expected output
  - Converts input to calculated output
  - Success if calculated matches expected
- Test Process
  - Add **one** set of test data
  - Test all the test data
  - Write code to make the tests pass

Performance

- Original code made three passes to get from orthogonal list to a final instruction sequence
- Substring comparisons potentially costly
- New code takes a single pass
- Code size down: 3114 bytes to 1348 bytes
- Run time down: 42000 cycles to 8500 cycles
- Easier to modify
- Still a 240 line function though
- There may be better implementation methods

Results

- Output from existing mouse from actual maze data matches new routine
- Development method assures correspondence with existing code
- No guarantee existing code is correct
- Untried in a maze

THANK YOU