CPU Instruction Set Manual

Overview
This manual describes the instruction set of the custom CPU simulated by the provided `CounterMachine` JavaScript class. It includes the syntax, description, and examples for each instruction.

Instruction Set

1. `LOAD`
Syntax:
```
LOAD rX, #imm
LOAD rX, [addr]
LOAD rX, [rY]
```
Description: Loads an immediate value, memory value, or indirectly addressed value into register `rX`.

Examples:
```assembly
LOAD r0, #10      ; Load immediate value 10 into register r0
LOAD r1, [50]     ; Load value from memory address 50 into register r1
LOAD r2, [r1]     ; Load value from memory address stored in register r1 into r2
```

---

2. `STORE`
Syntax:
```
STORE rX, [addr]
STORE rX, [rY]
```
Description: Stores the value from register `rX` into memory.

Examples:
```assembly
STORE r0, [100]    ; Store value from r0 into memory address 100
STORE r2, [r3]     ; Store value from r2 into memory address specified by r3
```

---

3. Stack Operations
- **`PUSH rX`**: Pushes the value of register `rX` onto the data stack.
- **`POP rX`**: Pops the top value from the data stack into register `rX`.

Examples:
```assembly
```

PUSH r1
POP r2
```

---

### 4. Memory Access via Stack Operations
- **`PEEK rX, [addr]`**: Reads memory at `[addr]` into register `rX`.
- **`POKE rX, [addr]`**: Writes value of register `rX` to memory at `[addr]`.

**Examples:**
```assembly
PEEK r1, [20]
POKE r2, [r1]
```

---

5. Arithmetic Operations
- **`ADD rX, rY`**: Adds register `rY` to `rX`.
- **`SUB rX, rY`**: Subtracts register `rY` from `rX`.
- **`MUL rX, rY`**: Multiplies `rX` by `rY`.
- **`DIV rX, rY`**: Integer division of `rX` by `rY`.

Examples:
```assembly
ADD r0, r1
SUB r2, r3
MUL r1, r2
DIV r3, r1
```

---

6. Control Flow Instructions
- **`JMP label`**: Unconditional jump to `label`.
- **`JZ rX, label`**: Jump to `label` if register `rX` is zero.
- **`JNZ rX, label`**: Jump to `label` if register `rX` is nonzero.

Examples:
```assembly
JMP loopStart
JZ r0, endLabel
JNZ r1, continueLoop
```

---

7. Subroutine and PL/0 Integration
- **`CALL label`**: Calls subroutine at `label`.
- **`PL0CALL programName`**: Calls compiled PL/0 program named `programName`.
- **`RET`**: Returns from subroutine or PL/0 call.
```

Examples:
```assembly
CALL myRoutine
PL0CALL matrixTest
RET
```

---

8. System Instructions
- **`HALT`**: Halts the CPU execution.

Example:
```assembly
HALT
```

Example Program
This short program demonstrates loading values, performing arithmetic, and halting:

```assembly
LOAD r0, #5
LOAD r1, #10
ADD r0, r1    ; r0 = 15
STORE r0, [100]
HALT
```

---

linear search

```
program LinearSearch;
var arrBase, n, target, i, current, found;
begin
  arrBase := 30;  // Base memory address of array
  n := 5;         // Number of elements
  target := 20;   // Element to search for

  i := 0;
  found := 0;

  while i < n do
  begin
    peek(current, arrBase + i);
    if current = target then
    begin
      found := 1;
      // Element found at index i
      i := n; // Exit loop
    end
    else
      i := i + 1;
  end;

  push found; // Push 1 if found, else 0
end.
```

Binary search

```
program BinarySearch;
var arrBase, left, right, mid, target, current, found;
begin
  arrBase := 30;    // Base memory address of sorted array
  left := 0;
  right := 4;       // last index (n - 1)
  target := 20;     // Element to search for
  found := 0;

  while left <= right do
  begin
    mid := (left + right) / 2;
    peek(current, arrBase + mid);

    if current = target then
    begin
      found := 1;
      left := right + 1; // Exit loop
    end
    else if current < target then
      left := mid + 1;
    else
      right := mid - 1;
  end;

  push found; // Push 1 if found, else 0
end.
```

Heapsort

```
program HeapSort;
var arrBase, n, i, largest, l, r, heapSize, temp, rootVal, childVal;

procedure swap(x, y);
var a, b;
begin
  peek(a, x);
  peek(b, y);
  poke(x, b);
  poke(y, a);
end;

procedure heapify(n, i);
begin
  largest := i;
  l := 2 * i + 1;
  r := 2 * i + 2;

  if l < n then
  begin
    peek(rootVal, arrBase + largest);
    peek(childVal, arrBase + l);
    if childVal > rootVal then
      largest := l;
  end;

  if r < n then
  begin
    peek(rootVal, arrBase + largest);
    peek(childVal, arrBase + r);
    if childVal > rootVal then
      largest := r;
  end;

  if largest <> i then
  begin
    call swap(arrBase + i, arrBase + largest);
    call heapify(n, largest);
  end;
end;

begin
  arrBase := 30;  // Base memory address of array
  n := 5;         // Number of elements
  heapSize := n;

  // Build heap
  i := (n / 2) - 1;
  while i >= 0 do
  begin
```

```
    call heapify(n, i);
    i := i - 1;
  end;

  // Extract elements from heap
  i := n - 1;
  while i > 0 do
  begin
    call swap(arrBase, arrBase + i);
    call heapify(i, 0);
    i := i - 1;
  end;
end.
```

6502 simulator and pl0 language listings

```
class CounterMachine {
  constructor(numCounters, memorySize = 256, dataStackSize = 256) {
    this.numCounters = numCounters;
    this.counters = new Array(numCounters).fill(0);
    this.memory = new Array(memorySize).fill(0);
    this.instructions = [];
    this.pointer = 0;
    this.callStack = [];
    this.dataStack = [];               // New data stack for PUSH/POP
    this.dataStackMax = dataStackSize;
    this.running = false;
    this.labelMap = {};
  }

  addInstruction(instruction) {
    this.instructions.push(instruction);
  }

  addInstructions(instructionsArray) {
    instructionsArray.forEach(line => this.addInstruction(line));
  }

  execute() {
    // 1) Build label map for jumps/calls
    this.buildLabelMap();
    // 2) Initialize pointer and set running flag
    this.pointer = 0;
    this.running = true;
    // 3) Fetch-decode-execute loop
    while (this.running && this.pointer < this.instructions.length) {
      let line = this.instructions[this.pointer].trim();
      // Skip labels
      if (line.endsWith(':')) {
        this.pointer++;
        continue;
```

```
    }
    let parts = line.split(/\s+/);
    let op = parts[0];
    let args = parts.slice(1);
    switch (op) {
      case 'LOAD': {
        // Format: LOAD rX, #imm  or  LOAD rX, [addr]
        let [regPart, valPart] = args;
        if (regPart.endsWith(',')) {
          regPart = regPart.slice(0, -1);
        }
        let regIndex = parseInt(regPart.replace('r', ''), 10);
        if (valPart.startsWith('#')) {
          // Immediate value
          let immediate = parseInt(valPart.slice(1), 10);
          this.counters[regIndex] = immediate;
        } else if (valPart.startsWith('[')) {
          // Memory read
          let addrStr = valPart.slice(1, -1);
          let addr;
          if (addrStr.toLowerCase().startsWith('r')) {
            // Indirect addressing: address is in register
            let addrRegIndex = parseInt(addrStr.replace('r', ''), 10);
            addr = this.counters[addrRegIndex];
          } else {
            addr = parseInt(addrStr, 10);
          }
          this.counters[regIndex] = this.memory[addr];
        }
        this.pointer++;
        break;
      }
      case 'STORE': {
        // Format: STORE rX, [addr]
        let [regPart, addrPart] = args;
        if (regPart.endsWith(',')) {
          regPart = regPart.slice(0, -1);
        }
        let xIndex = parseInt(regPart.replace('r', ''), 10);
        let addrStr = addrPart.slice(1, -1);
        let addr;
        if (addrStr.toLowerCase().startsWith('r')) {
          // Indirect addressing for store
          let addrRegIndex = parseInt(addrStr.replace('r', ''), 10);
          addr = this.counters[addrRegIndex];
        } else {
          addr = parseInt(addrStr, 10);
        }
        this.memory[addr] = this.counters[xIndex];
        this.pointer++;
        break;
      }
```

```
case 'PUSH': {
  // Format: PUSH rX
  let regIndex = parseInt(args[0].replace('r',''), 10);
  if (this.dataStack.length >= this.dataStackMax) {
    console.error("Data stack overflow");
    this.running = false;
    break;
  }
  this.dataStack.push(this.counters[regIndex]);
  this.pointer++;
  break;
}
case 'POP': {
  // Format: POP rX
  let regIndex = parseInt(args[0].replace('r',''), 10);
  if (this.dataStack.length === 0) {
    console.error("Data stack underflow");
    this.running = false;
    break;
  }
  this.counters[regIndex] = this.dataStack.pop();
  this.pointer++;
  break;
}
case 'PEEK': {
  // Format: PEEK rX, [addr]  (similar to LOAD)
  let [regPart, addrPart] = args;
  if (regPart.endsWith(',')) {
    regPart = regPart.slice(0, -1);
  }
  let regIndex = parseInt(regPart.replace('r', ''), 10);
  let addrStr = addrPart.slice(1, -1);
  let addr;
  if (addrStr.toLowerCase().startsWith('r')) {
    // address in register
    let addrRegIndex = parseInt(addrStr.replace('r', ''), 10);
    addr = this.counters[addrRegIndex];
  } else {
    addr = parseInt(addrStr, 10);
  }
  this.counters[regIndex] = this.memory[addr];
  this.pointer++;
  break;
}
case 'POKE': {
  // Format: POKE rX, [addr]  (similar to STORE)
  let [regPart, addrPart] = args;
  if (regPart.endsWith(',')) {
    regPart = regPart.slice(0, -1);
  }
  let xIndex = parseInt(regPart.replace('r', ''), 10);
  let addrStr = addrPart.slice(1, -1);
```

```
      let addr;
      if (addrStr.toLowerCase().startsWith('r')) {
        let addrRegIndex = parseInt(addrStr.replace('r', ''), 10);
        addr = this.counters[addrRegIndex];
      } else {
        addr = parseInt(addrStr, 10);
      }
      this.memory[addr] = this.counters[xIndex];
      this.pointer++;
      break;
    }
    case 'ADD': {
      // Format: ADD rX, rY  => rX = rX + rY
      let [rX, rY] = args;
      if (rX.endsWith(',')) rX = rX.slice(0, -1);
      let xIndex = parseInt(rX.replace('r', ''), 10);
      let yIndex = parseInt(rY.replace('r', ''), 10);
      this.counters[xIndex] += this.counters[yIndex];
      this.pointer++;
      break;
    }
    case 'SUB': {
      // Format: SUB rX, rY  => rX = rX - rY
      let [rX, rY] = args;
      if (rX.endsWith(',')) rX = rX.slice(0, -1);
      let xIndex = parseInt(rX.replace('r', ''), 10);
      let yIndex = parseInt(rY.replace('r', ''), 10);
      this.counters[xIndex] -= this.counters[yIndex];
      this.pointer++;
      break;
    }
    case 'MUL': {
      // Format: MUL rX, rY  => rX = rX * rY
      let [rX, rY] = args;
      if (rX.endsWith(',')) rX = rX.slice(0, -1);
      let xIndex = parseInt(rX.replace('r', ''), 10);
      let yIndex = parseInt(rY.replace('r', ''), 10);
      this.counters[xIndex] *= this.counters[yIndex];
      this.pointer++;
      break;
    }
    case 'DIV': {
      // Format: DIV rX, rY  => rX = rX / rY (integer division)
      let [rX, rY] = args;
      if (rX.endsWith(',')) rX = rX.slice(0, -1);
      let xIndex = parseInt(rX.replace('r', ''), 10);
      let yIndex = parseInt(rY.replace('r', ''), 10);
      if (this.counters[yIndex] === 0) {
        console.error("Division by zero");
        this.running = false;
        break;
      }
```

```
      this.counters[xIndex] = Math.floor(this.counters[xIndex] / this.counters[yIndex]);
      this.pointer++;
      break;
    }
    case 'JMP': {
      // Format: JMP label
      let label = args[0];
      this.pointer = this.labelMap[label];
      break;
    }
    case 'JZ': {
      // Format: JZ rX, label (jump if rX is zero)
      let [regPart, label] = args;
      if (regPart.endsWith(',')) {
        regPart = regPart.slice(0, -1);
      }
      let regIndex = parseInt(regPart.replace('r', ''), 10);
      if (this.counters[regIndex] === 0) {
        this.pointer = this.labelMap[label];
      } else {
        this.pointer++;
      }
      break;
    }
    case 'JNZ': {
      // Format: JNZ rX, label (jump if rX is not zero)
      let [regPart, label] = args;
      if (regPart.endsWith(',')) {
        regPart = regPart.slice(0, -1);
      }
      let regIndex = parseInt(regPart.replace('r', ''), 10);
      if (this.counters[regIndex] !== 0) {
        this.pointer = this.labelMap[label];
      } else {
        this.pointer++;
      }
      break;
    }
    case 'CALL': {
      // Normal subroutine call: push return address and jump to label
      let label = args[0];
      this.callStack.push(this.pointer + 1);
      this.pointer = this.labelMap[label];
      break;
    }
    case 'PL0CALL': {
      // Call a compiled PL/0 program by name
      let programName = args[0];
      if (programName.endsWith(',')) {
        programName = programName.slice(0, -1);
      }
      let newInstrs = PL0Programs[programName];
```

```
        if (!newInstrs) {
          console.error("No compiled PL/0 program named:", programName);
          this.running = false;
          break;
        }
        // Save current context on call stack
        this.callStack.push({
          instructions: this.instructions,
          labelMap: this.labelMap,
          returnPointer: this.pointer + 1
        });
        // Switch to the called program's instruction list and labels
        this.instructions = newInstrs;
        this.buildLabelMap();
        this.pointer = 0;
        break;
      }
      case 'RET': {
        // Return from subroutine or PL/0 call
        if (this.callStack.length > 0) {
          let top = this.callStack.pop();
          if (typeof top === 'number') {
            // Normal CALL return
            this.pointer = top;
          } else {
            // Return from PL0CALL
            this.instructions = top.instructions;
            this.labelMap = top.labelMap;
            this.pointer = top.returnPointer;
          }
        } else {
          // No context to return to: halt
          this.running = false;
        }
        break;
      }
      case 'HALT': {
        this.running = false;
        this.pointer++;
        break;
      }
      default: {
        console.error("Unknown instruction:", op, "at line", this.pointer, "-", line);
        this.running = false;
      }
    }
  }
}

buildLabelMap() {
  this.labelMap = {};
  for (let i = 0; i < this.instructions.length; i++) {
```

```
      let line = this.instructions[i].trim();
      if (line.endsWith(':')) {
        let labelName = line.slice(0, -1);
        this.labelMap[labelName] = i;
      }
    }
  }
}


// Tokenizer: splits source code into tokens (identifiers, numbers, symbols, keywords)
function tokenize(input) {
  // Remove single-line comments (// ...)
  input = input.replace(/\/\/.*$/gm, '');
  input = input
    .replace(/:=/g, ' := ')
    .replace(/\(/g, ' ( ')
    .replace(/\)/g, ' ) ')
    .replace(/,/g, ' , ')
    .replace(/;/g, ' ; ')
    .replace(/\./g, ' . ');
  const parts = input.split(/\s+/).filter(s => s.length > 0);
  const tokens = [];
  const keywords = ["program", "var", "begin", "end", "call", "if", "then", "while", "do", "odd",
"push", "pop", "peek", "poke"];
  const symbols = ["+", "-", "*", "/", ":=", "(", ")", ",", ";", ".", "="];
  for (let rt of parts) {
    if (/^\d+$/.test(rt)) {
      tokens.push({ type: "number", value: parseInt(rt, 10) });
    } else if (keywords.includes(rt.toLowerCase())) {
      tokens.push({ type: "keyword", value: rt.toLowerCase() });
    } else if (symbols.includes(rt)) {
      tokens.push({ type: "symbol", value: rt });
    } else if (/^[a-zA-Z]\w*$/.test(rt)) {
      tokens.push({ type: "ident", value: rt });
    } else {
      tokens.push({ type: "unknown", value: rt });
    }
  }
  return tokens;
}


class PL0Parser {
  constructor(tokens) {
    this.tokens = tokens;
    this.pos = 0;
    this.varTable = new Map();
    this.nextVarAddr = 0;
    this.labelCounter = 100;
    this.tempAddr = 254;
  }
```

```
currentToken() {
  return this.tokens[this.pos] || { type: "EOF", value: "" };
}
eat(expected) {
  const token = this.currentToken();
  if (token.value === expected || token.type === expected) {
    this.pos++;
    return token;
  } else {
    throw new Error(`Parse error: expected ${expected}, got ${token.value} at pos=${this.pos}`);
  }
}
newLabel() {
  const label = `label_${this.labelCounter}`;
  this.labelCounter++;
  return label;
}
newTemp() {
  const t = this.tempAddr;
  this.tempAddr--;
  return t;
}
declareVar(ident) {
  if (this.varTable.has(ident)) {
    throw new Error(`Variable '${ident}' already declared`);
  }
  const addr = this.nextVarAddr;
  this.nextVarAddr++;
  this.varTable.set(ident, addr);
  return addr;
}
getVarAddr(ident) {
  if (!this.varTable.has(ident)) {
    throw new Error(`Unknown variable '${ident}'`);
  }
  return this.varTable.get(ident);
}

// Grammar:
// program -> "program" ident ";" block "."
parseProgram() {
  this.eat("program");
  const progNameToken = this.currentToken();
  this.eat("ident");
  this.eat(";");
  const [blockAST, blockCode] = this.parseBlock();
  this.eat(".");
  const programAST = { type: "program", name: progNameToken.value, block: blockAST };
  return [programAST, blockCode];
}
```

```
// block -> varDecl? statement
parseBlock() {
  let varDecls = [];
  let codeVars = [];
  if (this.currentToken().value === "var") {
    [varDecls, codeVars] = this.parseVarDecl();
  }
  const [stmtAST, stmtCode] = this.parseStatement();
  const blockAST = { type: "block", varDecls, statement: stmtAST };
  // Combine variable initialization code (if any) with statement code
  const blockCode = codeVars.concat(stmtCode);
  return [blockAST, blockCode];
}

// varDecl -> "var" ident {"," ident} ";"
parseVarDecl() {
  this.eat("var");
  const decls = [];
  while (true) {
    const idToken = this.currentToken();
    this.eat("ident");
    const addr = this.declareVar(idToken.value);
    decls.push({ type: "varDecl", ident: idToken.value, addr: addr });
    if (this.currentToken().value === ",") {
      this.eat(",");
    } else {
      break;
    }
  }
  this.eat(";");
  // No specific code for var declarations (initial values default to 0)
  return [decls, []];
}

// statement -> assignment | callStmt | ifStmt | whileStmt | compoundStmt | pushStmt | popStmt |
peekStmt | pokeStmt | (empty)
parseStatement() {
  const tk = this.currentToken();
  if (tk.value === "call") {
    return this.parseCallStatement();
  } else if (tk.value === "if") {
    return this.parseIfStatement();
  } else if (tk.value === "while") {
    return this.parseWhileStatement();
  } else if (tk.value === "begin") {
    return this.parseCompoundStatement();
  } else if (tk.value === "push") {
    return this.parsePushStatement();
  } else if (tk.value === "pop") {
    return this.parsePopStatement();
  } else if (tk.value === "peek") {
    return this.parsePeekStatement();
```

```
  } else if (tk.value === "poke") {
    return this.parsePokeStatement();
  } else if (tk.type === "ident") {
    return this.parseAssignment();
  } else {
    // empty statement (possibly just a semicolon or end of block)
    return [{ type: "noop" }, []];
  }
}

// callStmt -> "call" ident ";"
parseCallStatement() {
  this.eat("call");
  const idToken = this.currentToken();
  this.eat("ident");
  this.eat(";");
  const ast = { type: "call", ident: idToken.value };
  const code = [`PL0CALL ${idToken.value}`];
  return [ast, code];
}

// assignment -> ident ":=" expression ";"
parseAssignment() {
  const idToken = this.currentToken();
  this.eat("ident");
  this.eat(":=");
  const [exprAST, exprCode] = this.parseExpression();
  this.eat(";");
  const addr = this.getVarAddr(idToken.value);
  const code = [
    ...exprCode,
    `STORE r0, [${addr}]`
  ];
  const ast = { type: "assign", ident: idToken.value, expr: exprAST };
  return [ast, code];
}

// pushStmt -> "push" ident ";"
parsePushStatement() {
  this.eat("push");
  const idToken = this.currentToken();
  this.eat("ident");
  this.eat(";");
  const addr = this.getVarAddr(idToken.value);
  // Load variable value into r0, then push it
  const code = [
    `LOAD r0, [${addr}]`,
    `PUSH r0`
  ];
  const ast = { type: "push", ident: idToken.value };
  return [ast, code];
}
```

```
// popStmt -> "pop" ident ";"
parsePopStatement() {
  this.eat("pop");
  const idToken = this.currentToken();
  this.eat("ident");
  this.eat(";");
  const addr = this.getVarAddr(idToken.value);
  // Pop top of stack into r0, then store r0 into variable
  const code = [
    `POP r0`,
    `STORE r0, [${addr}]`
  ];
  const ast = { type: "pop", ident: idToken.value };
  return [ast, code];
}

// peekStmt -> "peek(" ident "," ident ") ;"
parsePeekStatement() {
  this.eat("peek");
  this.eat("(");
  const destToken = this.currentToken();
  this.eat("ident");
  this.eat(",");
  const addrToken = this.currentToken();
  this.eat("ident");
  this.eat(")");
  this.eat(";");
  const destAddr = this.getVarAddr(destToken.value);
  const addrVarAddr = this.getVarAddr(addrToken.value);
  // Load the address value from addrVar into r0, then peek memory at that address into r1, then
store into dest
  const code = [
    `LOAD r0, [${addrVarAddr}]`,
    `PEEK r1, [r0]`,
    `STORE r1, [${destAddr}]`
  ];
  const ast = { type: "peek", dest: destToken.value, addr: addrToken.value };
  return [ast, code];
}

// pokeStmt -> "poke(" ident "," ident ") ;"
parsePokeStatement() {
  this.eat("poke");
  this.eat("(");
  const addrToken = this.currentToken();
  this.eat("ident");
  this.eat(",");
  const valToken = this.currentToken();
  this.eat("ident");
  this.eat(")");
  this.eat(";");
```

```
  const addrVarAddr = this.getVarAddr(addrToken.value);
  const valAddr = this.getVarAddr(valToken.value);
  // Load the target address from addrVar into r0, load the value into r1, then poke value into
memory at that address
  const code = [
    `LOAD r0, [${addrVarAddr}]`,
    `LOAD r1, [${valAddr}]`,
    `POKE r1, [r0]`
  ];
  const ast = { type: "poke", addr: addrToken.value, val: valToken.value };
  return [ast, code];
 }

 // ifStmt -> "if" expression "then" statement
 parseIfStatement() {
  this.eat("if");
  const [condAST, condCode] = this.parseExpression();
  this.eat("then");
  const [thenAST, thenCode] = this.parseStatement();
  const skipLabel = this.newLabel();
  const code = [
    ...condCode,
    `JZ r0, ${skipLabel}`,
    ...thenCode,
    `${skipLabel}:`
  ];
  const ast = { type: "if", condition: condAST, thenPart: thenAST };
  return [ast, code];
 }

 // whileStmt -> "while" expression "do" statement
 parseWhileStatement() {
  this.eat("while");
  const startLabel = this.newLabel();
  const exitLabel = this.newLabel();
  const loopStart = `${startLabel}:`;
  const [condAST, condCode] = this.parseExpression();
  this.eat("do");
  const [bodyAST, bodyCode] = this.parseStatement();
  const code = [
    loopStart,
    ...condCode,
    `JZ r0, ${exitLabel}`,
    ...bodyCode,
    `JMP ${startLabel}`,
    `${exitLabel}:`
  ];
  const ast = { type: "while", condition: condAST, body: bodyAST };
  return [ast, code];
 }

 // compoundStmt -> "begin" statement { ";" statement } "end"
```

```
parseCompoundStatement() {
this.eat("begin");
const stmts = [];
const codeAll = [];
// Loop until we reach "end"
while (this.currentToken().value !== "end") {
  const [stmtAST, stmtCode] = this.parseStatement();
  stmts.push(stmtAST);
  codeAll.push(...stmtCode);
  // If there's a semicolon, consume it (optional between statements)
  if (this.currentToken().value === ";") {
    this.eat(";");
  }
}
this.eat("end");
const ast = { type: "compound", statements: stmts };
return [ast, codeAll];
}

// expression -> term { (+|-) term }
parseExpression() {
  // Parse left term
  let [leftAST, leftCode] = this.parseTerm();
  // Handle + and - operators
  while (this.currentToken().value === "+" || this.currentToken().value === "-") {
    const opToken = this.currentToken().value;
    this.eat(opToken);
    const [rightAST, rightCode] = this.parseTerm();
    // Build AST node for binary operation
    const binAST = { type: "binop", op: opToken, left: leftAST, right: rightAST };
    // Reserve a temporary memory cell to save left operand
    const tempAddr = this.newTemp();
    const storeLeft = `STORE r0, [${tempAddr}]`;
    const loadLeftIntoR1 = `LOAD r1, [${tempAddr}]`;
    const opInstruction = opToken === "+" ? "ADD r0, r1" : "SUB r0, r1";
    const code = [
      ...leftCode,
      storeLeft,
      ...rightCode,
      loadLeftIntoR1,
      opInstruction
    ];
    leftAST = binAST;
    leftCode = code;
  }
  return [leftAST, leftCode];
}

// term -> factor { (*|/) factor }
parseTerm() {
  let [leftAST, leftCode] = this.parseFactor();
  while (this.currentToken().value === "*" || this.currentToken().value === "/") {
```

```
      const opToken = this.currentToken().value;
      this.eat(opToken);
      const [rightAST, rightCode] = this.parseFactor();
      const binAST = { type: "binop", op: opToken, left: leftAST, right: rightAST };
      const tempAddr = this.newTemp();
      const storeLeft = `STORE r0, [${tempAddr}]`;
      const loadLeftIntoR1 = `LOAD r1, [${tempAddr}]`;
      const opInstruction = opToken === "*" ? "MUL r0, r1" : "DIV r0, r1";
      const code = [
        ...leftCode,
        storeLeft,
        ...rightCode,
        loadLeftIntoR1,
        opInstruction
      ];
      leftAST = binAST;
      leftCode = code;
    }
    return [leftAST, leftCode];
  }

  // factor -> number | ident | "(" expression ")"
  parseFactor() {
    const tk = this.currentToken();
    if (tk.type === "number") {
      this.eat("number");
      const code = [`LOAD r0, #${tk.value}`];
      return [{ type: "num", value: tk.value }, code];
    } else if (tk.type === "ident") {
      this.eat("ident");
      const addr = this.getVarAddr(tk.value);
      const code = [`LOAD r0, [${addr}]`];
      return [{ type: "var", name: tk.value }, code];
    } else if (tk.value === "(") {
      this.eat("(");
      const [exprAST, exprCode] = this.parseExpression();
      this.eat(")");
      return [exprAST, exprCode];
    } else {
      throw new Error("Unexpected token in factor: " + JSON.stringify(tk));
    }
  }
}

// Global storage for compiled PL/0 programs
const PL0Programs = {};

// Compile PL/0 source code into CounterMachine instructions
function compilePL0(programText, baseAddr = 0) {
  const tokens = tokenize(programText);
  const parser = new PL0Parser(tokens);
  if (baseAddr !== 0) {
```

```
    parser.nextVarAddr = baseAddr;  // start allocating vars at a given memory address
  }
  const [ast, code] = parser.parseProgram();
  code.push("RET");  // ensure program returns to caller
  PL0Programs[ast.name] = code;
  return code;
}


// PL/0 subroutine to set a matrix element: writes val to memory at address (base + row*width +
col)
const setElementSource = `
program setElement;
var row, col, width, val, offset;
begin
  // Pop parameters from stack (expected order: val, width, col, row)
  pop val;
  pop width;
  pop col;
  pop row;
  // Compute linear offset = row * width + col
  offset := row * width + col;
  // Add base address 30 (start of matrix in memory)
  offset := offset + 30;
  // Write the value to memory at (base+offset)
  poke(offset, val);
end.
`;

// PL/0 subroutine to get a matrix element: reads memory at (base + row*width + col) and pushes
it
const getElementSource = `
program getElement;
var row, col, width, offset, value;
begin
  // Pop parameters from stack (expected order: width, col, row)
  pop width;
  pop col;
  pop row;
  // Compute linear offset = row * width + col
  offset := row * width + col;
  // Add base address 30
  offset := offset + 30;
  // Read the value from memory at (base+offset) into variable 'value'
  peek(value, offset);
  // Push the retrieved value onto the data stack as the result
  push value;
end.
`;

// Main PL/0 program to test matrix access
const matrixTestSource = `
```

```
program matrixTest;
var width, row, col, val;
begin
  // Initialize matrix dimensions and target indices
  width := 4;
  row := 2;
  col := 3;
  val := 99;
  // Push parameters and call setElement (writes val at matrix[row][col])
  push row;
  push col;
  push width;
  push val;
  call setElement;
  // Push parameters and call getElement (reads matrix[row][col] into stack)
  push row;
  push col;
  push width;
  call getElement;
  // (The returned value is now on top of the data stack)
end.
`;

// Compile the helper programs and the main program.
// We use a higher base address for matrixTest's variables to avoid overlap with others.
compilePL0(setElementSource);           // uses default baseAddr = 0
compilePL0(getElementSource);           // uses default baseAddr = 0
compilePL0(matrixTestSource, 20);       // place matrixTest variables at memory starting at addr
20

// Set up and run the machine
const machine = new CounterMachine(4, 256);
machine.addInstructions([
  "PL0CALL matrixTest",  // call the main test program
  "HALT"
]);
machine.execute();

// Output the memory state and data stack after execution
console.log("Final memory state (addresses 30-42):", machine.memory.slice(30, 42));
console.log("Final data stack:", machine.dataStack);
```