```
/**
 * BrailleCore - 中国现行盲文核心库
 * Chinese Braille Recognition and Translation Library
 */

// 盲文字符基址
const BRAILLE_BASE = 0x2800;

// 声母映射表
const INITIALS_MAP = {
  'b': [1, 2], 'p': [1, 2, 3, 4], 'm': [1, 3, 4], 'f': [1, 2, 4],
  'd': [1, 4, 5], 't': [2, 3, 4, 5], 'n': [1, 3, 4, 5], 'l': [1, 2, 3],
  'g': [1, 2, 4, 5], 'k': [1, 3], 'h': [1, 2, 5],
  'j': [1, 2, 4, 5], 'q': [1, 3], 'x': [1, 2, 5],
  'zh': [3, 4], 'ch': [1, 2, 3, 4, 5], 'sh': [1, 5, 6],
  'r': [2, 4, 5], 'z': [1, 3, 5, 6], 'c': [1, 4], 's': [2, 3, 4],
};

// 韵母映射表
const FINALS_MAP = {
  'a': [3, 5], 'o': [2, 6], 'e': [2, 6], 'i': [2, 4],
  'u': [1, 3, 6], 'v': [3, 4, 6], 'ü': [3, 4, 6],
  'ai': [2, 4, 6], 'ei': [2, 3, 4, 6], 'ao': [2, 3, 5],
  'ou': [1, 2, 3, 5, 6], 'an': [1, 2, 3, 6], 'en': [3, 5, 6],
  'ang': [2, 3, 6], 'eng': [3, 4, 5, 6], 'er': [1, 2, 3, 5],
  'ia': [1, 2, 4, 6], 'ie': [1, 5], 'iao': [3, 4, 5],
  'iu': [1, 2, 5, 6], 'ian': [1, 4, 6], 'in': [1, 2, 6],
  'iang': [1, 3, 4, 6], 'ing': [1, 6], 'iong': [1, 4, 5, 6],
  'ong': [2, 5, 6], 'ua': [1, 2, 3, 4, 5, 6], 'uai': [1, 3, 4, 5, 6],
  'uan': [1, 2, 4, 5, 6], 'un': [1, 2, 4, 5, 6], 'uang': [2, 3, 5, 6],
  'ui': [2, 4, 5, 6], 'uo': [1, 3, 5], 've': [2, 3, 4, 5, 6],
  'üe': [2, 3, 4, 5, 6], 'vn': [2, 5], 'ün': [2, 5],
};

// 声调映射表
const TONE_MAP = { 1: [1], 2: [2], 3: [3], 4: [2, 3] };

// 辅助函数
function dotsKey(dots) {
  return [...dots].sort((a, b) => a - b).join(',');
}

function dotsToBraille(dots) {
  let code = BRAILLE_BASE;
  for (const d of dots) {
    if (d === 1) code |= 0x01; if (d === 2) code |= 0x02;
    if (d === 3) code |= 0x04; if (d === 4) code |= 0x08;
    if (d === 5) code |= 0x10; if (d === 6) code |= 0x20;
  }
  return String.fromCharCode(code);
}

```

```
function brailleToDots(ch) {
  const code = ch.charCodeAt(0) - BRAILLE_BASE;
  const dots = [];
  if (code & 0x01) dots.push(1); if (code & 0x02) dots.push(2);
  if (code & 0x04) dots.push(3); if (code & 0x08) dots.push(4);
  if (code & 0x10) dots.push(5); if (code & 0x20) dots.push(6);
  return dots.length ? dots : null;
}

// 构建反向查找表
function buildReverseMaps() {
  const reverseInitials = {};
  const initialOrder = ['b', 'p', 'm', 'f', 'd', 't', 'n', 'l', 'g', 'k', 'h', 'zh', 'ch', 'sh', 'r', 'z', 'c', 's', 'j', 'q', 'x'];
  for (const k of initialOrder) {
    if (INITIALS_MAP[k]) reverseInitials[dotsKey(INITIALS_MAP[k])] = k;
  }
  const reverseFinals = {};
  const finalOrder = ['a', 'e', 'i', 'u', 'v', 'ai', 'ei', 'ao', 'ou', 'an', 'en', 'ang', 'eng', 'er', 'ia', 'ie', 'iao', 'iu', 'ian', 'in', 'iang', 'ing', 'iong', 'o
  for (const k of finalOrder) {
    if (FINALS_MAP[k]) reverseFinals[dotsKey(FINALS_MAP[k])] = k;
  }
  return { reverseInitials, reverseFinals };
}
const { reverseInitials, reverseFinals } = buildReverseMaps();

const INITIAL_LIST = ['zh', 'ch', 'sh', 'b', 'p', 'm', 'f', 'd', 't', 'n', 'l', 'g', 'k', 'h', 'j', 'q', 'x', 'r', 'z', 'c', 's'];

function parseSyllable(syllableWithTone) {
  const toneNum = parseInt(syllableWithTone.slice(-1));
  const tone = isNaN(toneNum) ? 0 : toneNum;
  const syllable = isNaN(toneNum) ? syllableWithTone : syllableWithTone.slice(0, -1);
  let initial = '', final = syllable;
  for (const ini of INITIAL_LIST) {
    if (syllable.startsWith(ini)) { initial = ini; final = syllable.slice(ini.length); break; }
  }
  if (!initial && syllable.startsWith('y')) {
    if (syllable === 'yu' || syllable.startsWith('yu')) final = syllable.replace(/^yu/, 'v');
    else if (syllable === 'yi') final = 'i';
    else final = syllable.replace(/^y/, 'i');
  }
  if (!initial && syllable.startsWith('w')) {
    if (syllable === 'wu') final = 'u';
    else final = syllable.replace(/^w/, 'u');
  }
  return { initial, final, tone };
}

function syllableToBraille(syllableWithTone, withTone = true) {
  const { initial, final: fin, tone } = parseSyllable(syllableWithTone);
  let result = '';
  if (initial && INITIALS_MAP[initial]) result += dotsToBraille(INITIALS_MAP[initial]);
  if (fin && FINALS_MAP[fin]) result += dotsToBraille(FINALS_MAP[fin]);
```

```
  else if (fin) for (const ch of fin) if (FINALS_MAP[ch]) result += dotsToBraille(FINALS_MAP[ch]);
  if (withTone && tone >= 1 && tone <= 4 && TONE_MAP[tone]) result += dotsToBraille(TONE_MAP[tone]);
  return result;
}

function isIFamilyFinal(final) {
  if (!final) return false;
  return final.startsWith('i') || final.startsWith('v') || final === 'ü' ||
    ['ia', 'ie', 'iao', 'iu', 'ian', 'in', 'iang', 'ing', 'iong', 've', 'vn'].includes(final);
}

function isValidCombination(initial, final) {
  if (!initial || !final) return false;
  const iOrVFamily = isIFamilyFinal(final) || final.startsWith('v') || final === 'ü';
  if (['j', 'q', 'x'].includes(initial)) return iOrVFamily;
  if (['g', 'k', 'h'].includes(initial)) return !iOrVFamily;
  return true;
}

function cellsToPinyin(cells) {
  const options = [];
  const ambiguousFinals = { '2,6': ['e', 'o'], '1,2,4,5,6': ['uan', 'un'] };
  for (const dots of cells) {
    const key = dotsKey(dots), opts = [];
    if (reverseFinals[key]) {
      if (ambiguousFinals[key]) {
        for (const fin of ambiguousFinals[key]) opts.push({ type: 'final', value: fin });
      } else {
        opts.push({ type: 'final', value: reverseFinals[key] });
      }
    }
    if (reverseInitials[key]) {
      const ambiguous = { '1,2,4,5': ['g', 'j'], '1,3': ['k', 'q'], '1,2,5': ['h', 'x'] };
      if (ambiguous[key]) for (const onset of ambiguous[key]) opts.push({ type: 'initial', value: onset });
      else opts.push({ type: 'initial', value: reverseInitials[key] });
    }
    opts.push({ type: 'unknown', value: null });
    options.push(opts);
  }
  return tryParse(options, 0, []);
}

function tryParse(options, idx, result) {
  if (idx >= options.length) return validateResult(result);
  for (const opt of options[idx]) {
    if (opt.type === 'unknown') continue;
    const newResult = addOption(result, opt);
    if (newResult) { const final = tryParse(options, idx + 1, newResult); if (final) return final; }
  }
  return null;
}

```

```
function addOption(currentResult, opt) {
  const newResult = currentResult.map(x => ({ ...x }));
  if (opt.type === 'initial') {
    if (newResult.length && !newResult[newResult.length-1].onset && newResult[newResult.length-1].rime) {
      const p = newResult[newResult.length-1].rime + opt.value;
      if (FINALS_MAP[p]) { newResult[newResult.length-1].rime = p; newResult[newResult.length-1].pinyin = p; return newResult
    }
    newResult.push({ onset: opt.value, rime: null, pinyin: null });
  } else if (opt.type === 'final') {
    if (newResult.length && newResult[newResult.length-1].onset && !newResult[newResult.length-1].rime) {
      if (isValidCombination(newResult[newResult.length-1].onset, opt.value)) {
        newResult[newResult.length-1].rime = opt.value;
        newResult[newResult.length-1].pinyin = newResult[newResult.length-1].onset + opt.value;
        return newResult;
      }
    }
    newResult.push({ onset: '', rime: opt.value, pinyin: opt.value });
  }
  return newResult;
}

function validateResult(result) {
  const pinyinParts = [], details = [];
  for (const item of result) {
    if (item.onset && item.rime) { pinyinParts.push(item.pinyin); details.push(item); }
    else if (item.onset) { pinyinParts.push(item.onset[0]); details.push({ onset: '', rime: item.onset[0], pinyin: item.onset[0] }); }
    else if (item.rime) { pinyinParts.push(item.rime); details.push(item); }
  }
  return pinyinParts.length ? { ok: true, pinyin: pinyinParts.join(' '), details } : null;
}

function parseCellsToPinyin(cells) {
  // 短序列用回溯，长序列用贪心
  if (cells.length <= 6) {
    const result = cellsToPinyin(cells);
    if (result) return { ok: true, pinyin: result.pinyin, details: result.details.map(d => ({ ...d, hanzi: '' })) };
  }
  const greedy = cellsToPinyinGreedy(cells);
  if (greedy) return { ok: true, pinyin: greedy.pinyin, details: greedy.details.map(d => ({ ...d, hanzi: '' })) };
  // 回退到回溯
  const result = cellsToPinyin(cells);
  if (!result) return { ok: false, pinyin: '', details: [], error: '无法找到有效的拼音组合' };
  return { ok: true, pinyin: result.pinyin, details: result.details.map(d => ({ ...d, hanzi: '' })) };
}

// ========== 贪心顺序解析器（支持声调、指示符、长序列） ==========

// 反向声调表
const reverseTones = {};
for (const [tone, dots] of Object.entries(TONE_MAP)) {
  reverseTones[dotsKey(dots)] = parseInt(tone);
}
```

```
209
210 // 指示符点位（单独出现时跳过）
211 const INDICATOR_KEYS = new Set(['4', '5', '6', '4,5', '4,6', '5,6', '4,5,6', '3,6']);
212
213 function classifyCell(dots) {
214   const key = dotsKey(dots);
215   const ambiguousFinals = { '2,6': ['e', 'o'], '1,2,4,5,6': ['uan', 'un'] };
216   const ambiguousInitials = { '1,2,4,5': ['g', 'j'], '1,3': ['k', 'q'], '1,2,5': ['h', 'x'] };
217
218   const result = { finals: [], initials: [], tone: null, isIndicator: false };
219
220   // 检查声调
221   if (reverseTones[key] !== undefined) {
222     result.tone = reverseTones[key];
223   }
224
225   // 检查韵母
226   if (reverseFinals[key]) {
227     if (ambiguousFinals[key]) {
228       result.finals = ambiguousFinals[key];
229     } else {
230       result.finals = [reverseFinals[key]];
231     }
232   }
233
234   // 检查声母
235   if (reverseInitials[key]) {
236     if (ambiguousInitials[key]) {
237       result.initials = ambiguousInitials[key];
238     } else {
239       result.initials = [reverseInitials[key]];
240     }
241   }
242
243   // 检查指示符
244   if (INDICATOR_KEYS.has(key) && result.finals.length === 0 && result.initials.length === 0) {
245     result.isIndicator = true;
246   }
247
248   return result;
249 }
250
251 function resolveAmbiguousInitial(initials, nextFinal) {
252   // g/j, k/q, h/x 的消歧：看后续韵母是否是 i族/v族
253   if (initials.length <= 1) return initials[0] || null;
254   const iOrV = nextFinal && (isIFamilyFinal(nextFinal) || nextFinal.startsWith('v') || nextFinal === 'ü');
255   // initials 顺序是 [g,j] / [k,q] / [h,x]
256   return iOrV ? initials[1] : initials[0];
257 }
258
259 function resolveAmbiguousFinal(finals, initial) {
260   // o/e 消歧：有声母时通常是 e（de, ge, he...），无声母时看具体情况
```

```
  if (finals.length <= 1) return finals[0] || null;
  if (finals.includes('e') && finals.includes('o')) {
    // 有声母 → 优先 e（更常见：de, ge, he, le, me, ne, se, ze, ce, re, te）
    // 无声母 → 优先 o（哦）但 e（饿）也可能，先返回 e
    // 特殊：b,p,m,f + o 是合法的（bo, po, mo, fo），其他声母 + o 不常见
    if (initial && ['b', 'p', 'm', 'f'].includes(initial)) return 'o';
    return 'e';
  }
  if (finals.includes('uan') && finals.includes('un')) {
    // 有声母 g,k,h,d,t,z,c,s,zh,ch,sh,r,l → uan 更常见
    // j,q,x → un（实际是 üan，但盲文写作 uan）
    if (initial && ['j', 'q', 'x'].includes(initial)) return 'un';
    return 'uan';
  }
  return finals[0];
}

function cellsToPinyinGreedy(cells) {
  const syllables = [];
  let i = 0;

  while (i < cells.length) {
    const cur = classifyCell(cells[i]);

    // 1. 纯声调方（前面没有匹配到音节，跳过孤立声调）
    if (cur.tone !== null && cur.finals.length === 0 && cur.initials.length === 0) {
      // 附加到前一个音节
      if (syllables.length > 0) syllables[syllables.length - 1].tone = cur.tone;
      i++;
      continue;
    }

    // 2. 指示符，跳过
    if (cur.isIndicator) {
      i++;
      continue;
    }

    // 3. 声母开头：尝试 声母 + 韵母 [+ 声调]
    if (cur.initials.length > 0) {
      // 看下一个方是否是韵母
      if (i + 1 < cells.length) {
        const next = classifyCell(cells[i + 1]);
        if (next.finals.length > 0) {
          // 先根据韵母消歧声母
          const fin = resolveAmbiguousFinal(next.finals, null);
          const ini = resolveAmbiguousInitial(cur.initials, fin);
          // 验证组合
          if (ini && fin && isValidCombination(ini, fin)) {
            const finalResolved = resolveAmbiguousFinal(next.finals, ini);
            const syl = { onset: ini, rime: finalResolved, pinyin: ini + finalResolved, tone: 0 };
            i += 2;
```

```
313           // 检查声调
314           if (i < cells.length) {
315             const toneCell = classifyCell(cells[i]);
316             if (toneCell.tone !== null && toneCell.finals.length === 0 && toneCell.initials.length === 0) {
317               syl.tone = toneCell.tone;
318               i++;
319             }
320           }
321           syllables.push(syl);
322           continue;
323         }
324         // 组合无效，尝试所有韵母×声母组合
325         let matched = false;
326         for (const f of next.finals) {
327           for (const ini2 of cur.initials) {
328             if (isValidCombination(ini2, f)) {
329               const syl = { onset: ini2, rime: f, pinyin: ini2 + f, tone: 0 };
330               i += 2;
331               if (i < cells.length) {
332                 const toneCell = classifyCell(cells[i]);
333                 if (toneCell.tone !== null && toneCell.finals.length === 0 && toneCell.initials.length === 0) {
334                   syl.tone = toneCell.tone;
335                   i++;
336                 }
337               }
338               syllables.push(syl);
339               matched = true;
340               break;
341             }
342           }
343           if (matched) break;
344         }
345         if (matched) continue;
346       }
347     }
348     // 声母后面不是韵母，可能是特殊情况（zh/ch/sh/z/c/s/r + i 的整体认读）
349     // 或者这个方其实应该当韵母用（如果它同时也是韵母）
350     if (cur.finals.length > 0) {
351       // 当作韵母处理
352       const fin = resolveAmbiguousFinal(cur.finals, '');
353       const syl = { onset: '', rime: fin, pinyin: fin, tone: 0 };
354       i++;
355       if (i < cells.length) {
356         const toneCell = classifyCell(cells[i]);
357         if (toneCell.tone !== null && toneCell.finals.length === 0 && toneCell.initials.length === 0) {
358           syl.tone = toneCell.tone;
359           i++;
360         }
361       }
362       syllables.push(syl);
363       continue;
364     }
```

```
      // 跳过无法匹配的声母
      i++;
      continue;
    }

    // 4. 韵母开头（零声母音节）：韵母 [+ 声调]
    if (cur.finals.length > 0) {
      const fin = resolveAmbiguousFinal(cur.finals, '');
      const syl = { onset: '', rime: fin, pinyin: fin, tone: 0 };
      i++;
      if (i < cells.length) {
        const toneCell = classifyCell(cells[i]);
        if (toneCell.tone !== null && toneCell.finals.length === 0 && toneCell.initials.length === 0) {
          syl.tone = toneCell.tone;
          i++;
        }
      }
      syllables.push(syl);
      continue;
    }

    // 5. 无法识别，跳过
    i++;
  }

  if (!syllables.length) return null;
  return {
    ok: true,
    pinyin: syllables.map(s => s.pinyin).join(' '),
    details: syllables.map(s => ({ onset: s.onset, rime: s.rime, pinyin: s.pinyin }))
  };
}

function tryParseAll(options, idx, result, collected, maxResults) {
  if (idx >= options.length) {
    const v = validateResult(result);
    if (v) collected.push(v);
    return;
  }
  for (const opt of options[idx]) {
    if (opt.type === 'unknown') continue;
    const newResult = addOption(result, opt);
    if (newResult) tryParseAll(options, idx + 1, newResult, collected, maxResults);
    if (collected.length >= maxResults) return;
  }
}

function cellsToPinyinAll(cells, maxResults = 10) {
  const options = [];
  const ambiguousFinals = { '2,6': ['e', 'o'], '1,2,4,5,6': ['uan', 'un'] };
  for (const dots of cells) {
    const key = dotsKey(dots), opts = [];
```

```
    if (reverseFinals[key]) {
      if (ambiguousFinals[key]) {
        for (const fin of ambiguousFinals[key]) opts.push({ type: 'final', value: fin });
      } else {
        opts.push({ type: 'final', value: reverseFinals[key] });
      }
    }
    if (reverseInitials[key]) {
      const ambiguous = { '1,2,4,5': ['g', 'j'], '1,3': ['k', 'q'], '1,2,5': ['h', 'x'] };
      if (ambiguous[key]) for (const onset of ambiguous[key]) opts.push({ type: 'initial', value: onset });
      else opts.push({ type: 'initial', value: reverseInitials[key] });
    }
    opts.push({ type: 'unknown', value: null });
    options.push(opts);
  }
  const collected = [];
  tryParseAll(options, 0, [], collected, maxResults);
  // 去重
  const seen = new Set();
  return collected.filter(r => {
    if (seen.has(r.pinyin)) return false;
    seen.add(r.pinyin);
    return true;
  });
}

function parseCellsToPinyinAll(cells, maxResults = 10) {
  // 长序列：贪心结果作为首选，回溯作为补充
  const candidates = [];
  const seen = new Set();

  // 贪心解析（始终尝试，对长序列是唯一可行方案）
  const greedy = cellsToPinyinGreedy(cells);
  if (greedy) {
    candidates.push({ pinyin: greedy.pinyin, details: greedy.details.map(d => ({ ...d, hanzi: '' })) });
    seen.add(greedy.pinyin);
  }

  // 短序列补充回溯结果
  if (cells.length <= 8) {
    const results = cellsToPinyinAll(cells, maxResults);
    for (const r of results) {
      if (!seen.has(r.pinyin)) {
        candidates.push({ pinyin: r.pinyin, details: r.details.map(d => ({ ...d, hanzi: '' })) });
        seen.add(r.pinyin);
      }
      if (candidates.length >= maxResults) break;
    }
  }

  if (!candidates.length) return { ok: false, candidates: [], error: '无法找到有效的拼音组合' };
  return { ok: true, candidates };
```

```
}

function validateSyllable(syllable) {
  if (!syllable) return { ok: false };
  const m = syllable.match(/^(\D+)(\d)?$/);
  if (!m) return { ok: false };
  const base = m[1], tone = m[2] || '';
  let initial = '', final = base;
  for (const ini of INITIAL_LIST) {
    if (base.startsWith(ini)) { initial = ini; final = base.slice(ini.length); break; }
  }
  if (!initial && base.startsWith('y')) final = base.replace(/^y/, 'i');
  if (!initial && base.startsWith('w')) final = base.replace(/^w/, 'u');
  if (!FINALS_MAP[final]) return { ok: false };
  if (['j', 'q', 'x'].includes(initial) && !isIFamilyFinal(final)) return { ok: false };
  return { ok: true };
}

function validatePinyin(pinyin) {
  const syllables = pinyin.trim().split(/\s+/).filter(Boolean);
  for (const s of syllables) if (!validateSyllable(s).ok) return { ok: false, isValid: false };
  return { ok: true, isValid: true };
}

function pinyinToHanzi(pinyin) {
  const map = { 'ma': '马', 'ma1': '妈', 'ma2': '麻', 'ma3': '马', 'ma4': '骂',
    'zhong': '中', 'zhong1': '中', 'zhong2': '种', 'zhong3': '种', 'zhong4': '众',
    'guo': '国', 'guo2': '国',
    'ni': '你', 'ni1': '尼', 'ni2': '尼', 'ni3': '你', 'ni4': '腻',
    'wo': '我', 'wo3': '我', 'wo4': '卧',
    'de': '的', 'de5': '地', 'de0': '得',
    'a': '啊', 'e': '饿', 'i': '一', 'o': '哦', 'u': '无', 'v': '鱼',
    'ai': '爱', 'ei': '诶', 'ao': '奥', 'ou': '欧',
    'an': '安', 'en': '恩', 'ang': '昂', 'eng': '鞥', 'er': '而',
    'ia': '呀', 'ie': '耶', 'iao': '要', 'iu': '有', 'ian': '烟', 'in': '因',
    'iang': '阳', 'ing': '应', 'iong': '用',
    'ua': '挖', 'uai': '外', 'uan': '万', 'uang': '王', 'ui': '回', 'uo': '我',
    've': '约', 'vn': '云',
    'ba': '八', 'bai': '白', 'ban': '半', 'bang': '帮', 'bao': '包',
    'bei': '被', 'ben': '本', 'beng': '蹦', 'bi': '比', 'bian': '边', 'biao': '表',
    'bie': '别', 'bin': '宾', 'bing': '并', 'bo': '波', 'bu': '不',
    'da': '大', 'dai': '带', 'dan': '单', 'dang': '当', 'dao': '到',
    'de': '的', 'dei': '得', 'deng': '等', 'di': '地', 'dian': '点',
    'diao': '调', 'die': '爹', 'ding': '定', 'diu': '丢', 'dong': '东',
    'dou': '都', 'du': '都', 'duan': '短', 'dui': '对', 'dun': '顿', 'duo': '多',
    'fa': '发', 'fan': '反', 'fang': '方', 'fei': '非', 'fen': '分', 'feng': '风',
    'fo': '佛', 'fou': '否', 'fu': '服',
    'ga': '嘎', 'gai': '改', 'gan': '干', 'gang': '刚', 'gao': '高',
    'ge': '个', 'gei': '给', 'gen': '根', 'geng': '更', 'gong': '工',
    'gou': '狗', 'gu': '古', 'gua': '挂', 'guai': '怪', 'guan': '关',
    'guang': '光', 'gui': '归', 'gun': '滚', 'guo': '国',
    'ha': '哈', 'hai': '海', 'han': '含', 'hang': '行', 'hao': '好',
```

```
'he': '和', 'hei': '黑', 'hen': '很', 'heng': '恒', 'hong': '红',
'hou': '后', 'hu': '胡', 'hua': '花', 'huai': '怀', 'huan': '换',
'huang': '黄', 'hui': '回', 'hun': '混', 'huo': '活',
'ji': '机', 'jia': '家', 'jian': '间', 'jiang': '江', 'jiao': '教',
'jie': '接', 'jin': '金', 'jing': '经', 'jiong': '窘', 'jiu': '九',
'ju': '局', 'juan': '卷', 'jue': '决', 'jun': '军',
'ka': '卡', 'kai': '开', 'kan': '看', 'kang': '抗', 'kao': '考',
'ke': '可', 'ken': '肯', 'keng': '坑', 'kong': '空', 'kou': '口',
'ku': '苦', 'kua': '跨', 'kuai': '快', 'kuan': '宽', 'kuang': '矿',
'kui': '亏', 'kun': '困', 'kuo': '阔',
'la': '拉', 'lai': '来', 'lan': '蓝', 'lang': '浪', 'lao': '老',
'le': '了', 'lei': '类', 'leng': '冷', 'li': '里', 'lia': '俩',
'lian': '连', 'liang': '两', 'liao': '料', 'lie': '列', 'lin': '林',
'ling': '领', 'liu': '六', 'long': '龙', 'lou': '楼',
'lu': '路', 'lv': '绿', 'luan': '乱', 'lun': '轮', 'luo': '罗',
'mei': '没', 'men': '门', 'meng': '梦', 'mi': '米', 'mian': '面',
'miao': '秒', 'mie': '咩', 'min': '民', 'ming': '明', 'miu': '谬',
'mo': '摸', 'mou': '某', 'mu': '木',
'na': '那', 'nai': '奶', 'nan': '南', 'nang': '囊', 'nao': '脑',
'ne': '呢', 'nei': '内', 'nen': '嫩', 'neng': '能', 'nian': '年',
'niang': '娘', 'niao': '鸟', 'nie': '捏', 'nin': '您', 'ning': '宁',
'niu': '牛', 'nong': '农', 'nu': '奴', 'nv': '女', 'nuan': '暖',
'nue': '虐', 'nuo': '诺',
'pa': '爬', 'pai': '排', 'pan': '盘', 'pang': '旁', 'pao': '跑',
'pei': '配', 'pen': '盆', 'peng': '碰', 'pi': '皮', 'pian': '片',
'piao': '票', 'pie': '瞥', 'pin': '拼', 'ping': '平', 'po': '破',
'pou': '剖', 'pu': '普',
'qi': '七', 'qia': '恰', 'qian': '千', 'qiang': '强', 'qiao': '桥',
'qie': '切', 'qin': '亲', 'qing': '情', 'qiong': '穷', 'qiu': '求',
'qu': '去', 'quan': '全', 'que': '确', 'qun': '群',
'ran': '然', 'rang': '让', 'rao': '绕', 're': '热', 'ren': '人',
'reng': '仍', 'ri': '日', 'rong': '容', 'rou': '肉', 'ru': '如',
'ruan': '软', 'rui': '锐', 'run': '润', 'ruo': '若',
'sa': '撒', 'sai': '赛', 'san': '三', 'sang': '桑', 'sao': '扫',
'se': '色', 'sen': '森', 'seng': '僧', 'sha': '沙', 'shan': '山',
'shang': '上', 'shao': '少', 'she': '社', 'shen': '神', 'sheng': '生',
'shi': '是', 'shou': '收', 'shu': '书', 'shua': '刷', 'shuai': '帅',
'shuan': '栓', 'shuang': '双', 'shui': '水', 'shun': '顺', 'shuo': '说',
'si': '四', 'song': '送', 'sou': '搜', 'su': '苏', 'suan': '算',
'sui': '随', 'sun': '孙', 'suo': '所',
'ta': '他', 'tai': '太', 'tan': '谈', 'tang': '汤', 'tao': '桃',
'te': '特', 'teng': '疼', 'ti': '体', 'tian': '天', 'tiao': '条',
'tie': '铁', 'ting': '听', 'tong': '同', 'tou': '头', 'tu': '图',
'tuan': '团', 'tui': '退', 'tun': '屯', 'tuo': '脱',
'wa': '瓦', 'wai': '外', 'wan': '万', 'wang': '王', 'wei': '为',
'wen': '文', 'weng': '翁', 'wo': '我', 'wu': '无',
'xi': '西', 'xia': '下', 'xian': '先', 'xiang': '想', 'xiao': '小',
'xie': '写', 'xin': '新', 'xing': '行', 'xiong': '熊', 'xiu': '修',
'xu': '需', 'xuan': '选', 'xue': '学', 'xun': '寻',
'ya': '亚', 'yan': '严', 'yang': '阳', 'yao': '要', 'ye': '也',
'yi': '一', 'yin': '音', 'ying': '应', 'yo': '哟', 'yong': '用',
'you': '有', 'yu': '于', 'yuan': '元', 'yue': '月', 'yun': '云',
```

```
    'za': '杂', 'zai': '在', 'zan': '赞', 'zang': '脏', 'zao': '早',
    'ze': '则', 'zei': '贼', 'zen': '怎', 'zeng': '增', 'zha': '扎',
    'zhai': '摘', 'zhan': '战', 'zhang': '张', 'zhao': '找', 'zhe': '这',
    'zhen': '真', 'zheng': '正', 'zhi': '知', 'zhou': '周', 'zhu': '主',
    'zhua': '抓', 'zhuai': '拽', 'zhuan': '专', 'zhuang': '装',
    'zhui': '追', 'zhun': '准', 'zhuo': '桌', 'zi': '子', 'zong': '总',
    'zou': '走', 'zu': '组', 'zuan': '钻', 'zui': '最', 'zun': '尊', 'zuo': '坐',
  };
  const syllables = pinyin.trim().split(/\s+/).filter(Boolean);
  let hanzi = '';
  for (const s of syllables) {
    const char = map[s] || map[s.replace(/\d$/, '')] || '?';
    hanzi += char;
  }
  return { ok: hanzi !== '', hanzi };
}

const BrailleCore = {
  dotsToBraille, brailleToDots, dotsKey,
  brailleCharToDots: brailleToDots,
  parseCellsInput: (input) => {
    const cells = [], parts = input.trim().split(/\s+/);
    for (const part of parts) {
      const dots = part.replace(/[\[\]]/g, '').split(/[,\s]+/).map(d => parseInt(d)).filter(d => !isNaN(d));
      if (dots.length) cells.push(dots);
    }
    return cells.length ? { ok: true, cells } : { ok: false };
  },
  parseSyllable, syllableToBraille, parseCellsToPinyin, parseCellsToPinyinAll, pinyinToHanzi,
  validateSyllable, validatePinyin, validateSequence: p => p.trim() ? { ok: true } : { ok: false },
  INITIALS_MAP, FINALS_MAP, TONE_MAP,
};

if (typeof module !== 'undefined') module.exports = BrailleCore;
```