Concatenated Source Code

Generated on 2025-09-14T05:06:13.453Z

Input directory: /Users/administrador/P/opensource/ripple/ripple/packages/ripple

src/compiler/phases/1-parse/index.js

```
import * as acorn from 'acorn';
import { tsPlugin } from 'acorn-typescript';
import { parse_style } from './style.js';

const parser = acorn.Parser.extend(tsPlugin({ allowSatisfies: true }), RipplePlugin());

function RipplePlugin(config) {
 return (Parser) => {
  const original = acorn.Parser.prototype;
  const tt = Parser.tokTypes || acorn.tokTypes;
  const tc = Parser.tokContexts || acorn.tokContexts;

  class RippleParser extends Parser {
   #path = [];

   shouldParseExportStatement() {
    if (super.shouldParseExportStatement()) {
     return true;
    }
    if (this.value === 'component' || this.value === 'fragment') {
     return true;
    }
    return this.type.keyword === 'var';
   }

   jsx_parseExpressionContainer() {
    const tok = this.acornTypeScript.tokTypes;
    let node = this.startNode();
    this.next();

    if (this.type === tok.at) {
     this.next();

     if (this.value === 'fragment') {
      node.decorator = 'fragment';
      this.next();
     } else {
      throw new Error(`Invalid syntax @` + this.value);
     }
    }

    node.expression =
     this.type === tt.braceR ? this.jsx_parseEmptyExpression() : this.parseExpression();
    this.expect(tt.braceR);
    return this.finishNode(node, 'JSXExpressionContainer');
   }

   jsx_parseTupleContainer() {
    var t = this.startNode();
```

```
     return (
      this.next(),
      (t.expression =
       this.type === tt.bracketR ? this.jsx_parseEmptyExpression() : this.parseExpression()),
      this.expect(tt.bracketR),
      this.finishNode(t, 'JSXExpressionContainer')
     );
    }

    jsx_parseAttribute() {
     let node = this.startNode();
     if (this.eat(tt.braceL)) {
      if (this.lookahead().type === tt.ellipsis) {
       this.expect(tt.ellipsis);
       node.argument = this.parseMaybeAssign();
       this.expect(tt.braceR);
       return this.finishNode(node, 'SpreadAttribute');
      } else {
       const id = this.parseIdentNode();
       this.finishNode(id, 'Identifier');
       node.name = id;
       node.value = id;
       this.next();
       this.expect(tt.braceR);
       return this.finishNode(node, 'Attribute');
      }
     }
     node.name = this.jsx_parseNamespacedName();
     node.value = this.eat(tt.eq) ? this.jsx_parseAttributeValue() : null;
     return this.finishNode(node, 'JSXAttribute');
    }

    jsx_parseAttributeValue() {
     const tok = this.acornTypeScript.tokTypes;

     switch (this.type) {
      case tt.braceL:
       var t = this.jsx_parseExpressionContainer();
       return (
        'JSXEmptyExpression' === t.expression.type &&
         this.raise(t.start, 'attributes must only be assigned a non-empty expression'),
        t
       );
      // case tt.bracketL:
      //  var t = this.jsx_parseTupleContainer();
      //  return (
      //      'JSXEmptyExpression' === t.expression.type &&
      //          this.raise(t.start, 'attributes must only be assigned a non-empty expression'),
      //      t
      //  );
      case tok.jsxTagStart:
      case tt.string:
       return this.parseExprAtom();
      default:
       this.raise(this.start, 'value should be either an expression or a quoted text');
     }
    }

    parseTryStatement(node) {
     this.next();
```

```
    node.block = this.parseBlock();
    node.handler = null;
    if (this.type === tt._catch) {
     var clause = this.startNode();
     this.next();
     if (this.eat(tt.parenL)) {
      clause.param = this.parseCatchClauseParam();
     } else {
      if (this.options.ecmaVersion < 10) {
       this.unexpected();
      }
      clause.param = null;
      this.enterScope(0);
     }
     clause.body = this.parseBlock(false);
     this.exitScope();
     node.handler = this.finishNode(clause, 'CatchClause');
    }
    node.finalizer = this.eat(tt._finally) ? this.parseBlock() : null;

    if (this.value === 'async') {
     this.next();
     node.async = this.parseBlock();
    } else {
     node.async = null;
    }

    if (!node.handler && !node.finalizer && !node.async) {
     this.raise(node.start, 'Missing catch or finally clause');
    }
    return this.finishNode(node, 'TryStatement');
   }

   jsx_readToken() {
    let out = '',
     chunkStart = this.pos;
    const tok = this.acornTypeScript.tokTypes;

    for (;;) {
     if (this.pos >= this.input.length) this.raise(this.start, 'Unterminated JSX contents');
     let ch = this.input.charCodeAt(this.pos);

     switch (ch) {
      case 60: // '<'
      case 123: // '{'
       if (ch === 60 && this.exprAllowed) {
        ++this.pos;
        return this.finishToken(tok.jsxTagStart);
       }
       if (ch === 123 && this.exprAllowed) {
        return this.getTokenFromCode(ch);
       }
       debugger;
       throw new Error('TODO: Invalid syntax');

      case 47: // '/'
       // Check if this is a comment (// or /*)
       if (this.input.charCodeAt(this.pos + 1) === 47) {
        // '//'
        // Line comment - handle it properly
```

```
        const commentStart = this.pos;
        const startLoc = this.curPosition();
        this.pos += 2;

        let commentText = '';
        while (this.pos < this.input.length) {
         const nextCh = this.input.charCodeAt(this.pos);
         if (acorn.isNewLine(nextCh)) break;
         commentText += this.input[this.pos];
         this.pos++;
        }

        const commentEnd = this.pos;
        const endLoc = this.curPosition();

        // Call onComment if it exists
        if (this.options.onComment) {
         this.options.onComment(
          false,
          commentText,
          commentStart,
          commentEnd,
          startLoc,
          endLoc
         );
        }

        // Continue processing from current position
        break;
       } else if (this.input.charCodeAt(this.pos + 1) === 42) {
        // '/*'
        // Block comment - handle it properly
        const commentStart = this.pos;
        const startLoc = this.curPosition();
        this.pos += 2;

        let commentText = '';
        while (this.pos < this.input.length - 1) {
         if (
          this.input.charCodeAt(this.pos) === 42 &&
          this.input.charCodeAt(this.pos + 1) === 47
         ) {
          this.pos += 2;
          break;
         }
         commentText += this.input[this.pos];
         this.pos++;
        }

        const commentEnd = this.pos;
        const endLoc = this.curPosition();

        // Call onComment if it exists
        if (this.options.onComment) {
         this.options.onComment(
          true,
          commentText,
          commentStart,
          commentEnd,
          startLoc,
```

```
          endLoc
         );
        }

        // Continue processing from current position
        break;
       }
       // If not a comment, fall through to default case
       this.context.push(tc.b_stat);
       this.exprAllowed = true;
       return original.readToken.call(this, ch);

      case 38: // '&'
       out += this.input.slice(chunkStart, this.pos);
       out += this.jsx_readEntity();
       chunkStart = this.pos;
       break;

      case 62: // '>'
      case 125: {
       // '}'
       if (
        ch === 125 &&
        (this.#path.at(-1).type === 'Component' || this.#path.at(-1).type === 'Fragment')
       ) {
        return original.readToken.call(this, ch);
       }
       this.raise(
        this.pos,
        'Unexpected token `' +
         this.input[this.pos] +
         '`. Did you mean `' +
         (ch === 62 ? '&gt;' : '&rbrace;') +
         '` or ' +
         '`{"' +
         this.input[this.pos] +
         '"}' +
         '`?'
       );
      }

      default:
       if (acorn.isNewLine(ch)) {
        out += this.input.slice(chunkStart, this.pos);
        out += this.jsx_readNewLine(true);
        chunkStart = this.pos;
       } else if (ch === 32) {
        ++this.pos;
       } else {
        this.context.push(tc.b_stat);
        this.exprAllowed = true;
        return original.readToken.call(this, ch);
       }
     }
    }
   }

   parseElement() {
    const tok = this.acornTypeScript.tokTypes;
    // Adjust the start so we capture the `<` as part of the element
```

```
    const prev_pos = this.pos;
    this.pos = this.start - 1;
    const position = this.curPosition();
    this.pos = prev_pos;

    const element = this.startNode();
    element.start = position.index;
    element.loc.start = position;
    element.type = 'Element';
    this.#path.push(element);
    element.children = [];
    const open = this.jsx_parseOpeningElementAt();
    for (const attr of open.attributes) {
     if (attr.type === 'JSXAttribute') {
      attr.type = 'Attribute';
      if (attr.name.type === 'JSXIdentifier') {
       attr.name.type = 'Identifier';
      }
      if (attr.value.type === 'JSXExpressionContainer') {
       attr.value = attr.value.expression;
      }
     }
    }
    if (open.name.type === 'JSXIdentifier') {
     open.name.type = 'Identifier';
    }
    element.id = open.name;
    element.id.type = 'Identifier';
    element.attributes = open.attributes;
    element.selfClosing = open.selfClosing;
    element.metadata = {};

    if (element.selfClosing) {
     this.#path.pop();
     if (this.type !== tok.jsxTagStart) {
      // Eat the closing `/>`
      this.pos--;
      this.next();
     }
    } else {
     if (open.name.name === 'style') {
      const start = this.start;
      const input = this.input.slice(start);
      const end = input.indexOf('</style>');
      const content = input.slice(0, end);

      const component = this.#path.findLast((n) => n.type === 'Component');
      if (component.css !== null) {
       throw new Error('Components can only have one style tag');
      }
      component.css = parse_style(content);

      this.pos = start + end + 1;
      this.type = tok.jsxTagStart;
      this.next();
      if (this.value === '/') {
       this.next();
       this.jsx_parseElementName();
       this.exprAllowed = true;
       this.#path.pop();
```

```
        this.next();
       }
       return null;
      } else {
       this.parseTemplateBody(element.children);
      }
     }

     this.finishNode(element, 'Element');
     return element;
    }

    parseTemplateBody(body) {
     var inside_func =
      this.context.some((n) => n.token === 'function') || this.scopeStack.length > 1;

     if (!inside_func) {
      if (this.type.label === 'return') {
       throw new Error('`return` statements are not allowed in components');
      }
      if (this.type.label === 'continue') {
       throw new Error('`continue` statements are not allowed in components');
      }
      if (this.type.label === 'break') {
       throw new Error('`break` statements are not allowed in components');
      }
     }

     if (this.type.label === '{') {
      const node = this.jsx_parseExpressionContainer();
      node.type = node.decorator === 'fragment' ? 'RenderFragment' : 'Text';
      if (node.decorator === 'fragment' && node.expression.type !== 'CallExpression') {
       throw new Error('{@fragment} must be a function call');
      }
      body.push(node);
     } else if (this.type.label === '}') {
      return;
     } else if (this.type.label === 'jsxTagStart') {
      this.next();
      if (this.value === '/') {
       this.next();
       this.jsx_parseElementName();
       this.exprAllowed = true;
       this.#path.pop();
       this.next();
       return;
      }
      const node = this.parseElement();
      if (node !== null) {
       body.push(node);
      }
     } else {
      const node = this.parseStatement(null);
      body.push(node);
     }
     this.parseTemplateBody(body);
    }

    parseStatement(context, topLevel, exports) {
     const tok = this.acornTypeScript.tokContexts;
```

```
    if (
     context !== 'for' &&
     context !== 'if' &&
     this.context.at(-1) === tc.b_stat &&
     this.type === tt.braceL &&
     this.context.some((c) => c === tok.tc_expr)
    ) {
     this.next();
     const node = this.jsx_parseExpressionContainer();
     node.type = 'Text';
     this.next();
     this.context.pop();
     this.context.pop();
     return node;
    }

    if (this.value === 'component') {
     const node = this.startNode();
     node.type = 'Component';
     node.css = null;
     this.next();
     this.enterScope(0);
     node.id = this.parseIdent();
     this.parseFunctionParams(node);
     this.eat(tt.braceL);
     node.body = [];
     this.#path.push(node);

     this.parseTemplateBody(node.body);

     this.#path.pop();
     this.exitScope();

     this.next();
     this.finishNode(node, 'Component');
     this.awaitPos = 0;

     return node;
    }

    if (this.value === 'fragment') {
     const node = this.startNode();
     node.type = 'Fragment';
     this.next();
     this.enterScope(0);
     node.id = this.parseIdent();
     this.parseFunctionParams(node);
     this.eat(tt.braceL);
     node.body = [];
     this.#path.push(node);

     this.parseTemplateBody(node.body);

     this.#path.pop();
     this.exitScope();

     this.finishNode(node, 'Fragment');
     this.next();
     this.awaitPos = 0;
```

```
      return node;
     }

     return super.parseStatement(context, topLevel, exports);
    }

    parseBlock(createNewLexicalScope, node, exitStrict) {
     const parent = this.#path.at(-1);

     if (
      parent?.type === 'Component' ||
      parent?.type === 'Fragment' ||
      parent?.type === 'Element'
     ) {
      if (createNewLexicalScope === void 0) createNewLexicalScope = true;
      if (node === void 0) node = this.startNode();

      node.body = [];
      this.expect(tt.braceL);
      if (createNewLexicalScope) {
       this.enterScope(0);
      }
      this.parseTemplateBody(node.body);

      if (exitStrict) {
       this.strict = false;
      }
      this.exprAllowed = true;

      this.next();
      if (createNewLexicalScope) {
       this.exitScope();
      }
      return this.finishNode(node, 'BlockStatement');
     }

     return super.parseBlock(createNewLexicalScope, node, exitStrict);
    }
   }

   return RippleParser;
  };
}

export function parse(source) {
 const comments = [];
 let ast;

 try {
  ast = parser.parse(source, {
   sourceType: 'module',
   ecmaVersion: 13,
   locations: true,
   onComment: (block, text, start, end, startLoc, endLoc) => {
    comments.push({
     type: block ? 'Block' : 'Line',
     value: text,
     start,
     end,
```

```
      loc: {
       start: startLoc,
       end: endLoc
      }
     });
    }
   });
  } catch (e) {
   throw e;
  }

  ast.comments = comments;
  return ast;
}
```

src/compiler/phases/1-parse/style.js

```
import { hash } from "../../utils.js";

const REGEX_COMMENT_CLOSE = /\*\//;
const REGEX_HTML_COMMENT_CLOSE = /-->/;
const REGEX_PERCENTAGE = /^\d+(\.\d+)?%/;
const REGEX_COMBINATOR = /^(\+|~|>|\|\|)/;
const REGEX_VALID_IDENTIFIER_CHAR = /[a-zA-Z0-9_-]/;
const REGEX_LEADING_HYPHEN_OR_DIGIT = /-?\d/;
const REGEX_WHITESPACE_OR_COLON = /[\s:]/;
const REGEX_NTH_OF =
 /^(even|odd|\+?(\d+|\d*n(\s*[+-]\s*\d+)?)|-\d*n(\s*\+\s*\d+))((?=\s*[,)])|\s+of\s+)/;

const regex_whitespace = /\s/;

class Parser {
 index = 0;

 constructor(template, loose) {
  if (typeof template !== 'string') {
   throw new TypeError('Template must be a string');
  }

  this.loose = loose;
  this.template_untrimmed = template;
  this.template = template.trimEnd();
 }

 match(str) {
  const length = str.length;
  if (length === 1) {
   // more performant than slicing
   return this.template[this.index] === str;
  }

  return this.template.slice(this.index, this.index + length) === str;
 }

 eat(str, required = false, required_in_loose = true) {
  if (this.match(str)) {
   this.index += str.length;
   return true;
```

```
    }

    if (required && (!this.loose || required_in_loose)) {
      throw new Error(`Expected ${str}`);
    }

    return false;
  }

  match_regex(pattern) {
    const match = pattern.exec(this.template.slice(this.index));
    if (!match || match.index !== 0) return null;

    return match[0];
  }

  read(pattern) {
    const result = this.match_regex(pattern);
    if (result) this.index += result.length;
    return result;
  }

  allow_whitespace() {
    while (this.index < this.template.length && regex_whitespace.test(this.template[this.index])) {
      this.index++;
    }
  }

  read_until(pattern) {
    if (this.index >= this.template.length) {
      if (this.loose) return '';
      throw new Error('Unexpected end of input');
    }

    const start = this.index;
    const match = pattern.exec(this.template.slice(start));

    if (match) {
      this.index = start + match.index;
      return this.template.slice(start, this.index);
    }

    this.index = this.template.length;
    return this.template.slice(start);
  }
}

export function parse_style(content) {
  const parser = new Parser(content, false);

  return {
    source: content,
    hash: `ripple-${hash(content)}`,
    type: 'StyleSheet',
    body: read_body(parser)
  };
}

function allow_comment_or_whitespace(parser) {
  parser.allow_whitespace();
```

```
  while (parser.match('/*') || parser.match('<!--')) {
   if (parser.eat('/*')) {
    parser.read_until(REGEX_COMMENT_CLOSE);
    parser.eat('*/', true);
   }

   if (parser.eat('<!--')) {
    parser.read_until(REGEX_HTML_COMMENT_CLOSE);
    parser.eat('-->', true);
   }

   parser.allow_whitespace();
  }
}

function read_body(parser) {
 const children = [];

 while (parser.index < parser.template.length) {
  allow_comment_or_whitespace(parser);

  if (parser.match('@')) {
   children.push(read_at_rule(parser));
  } else {
   children.push(read_rule(parser));
  }
 }

 return children;
}

function read_at_rule(parser) {
 debugger;
}

function read_rule(parser) {
 const start = parser.index;

 return {
  type: 'Rule',
  prelude: read_selector_list(parser),
  block: read_block(parser),
  start,
  end: parser.index,
  metadata: {
   parent_rule: null,
   has_local_selectors: false,
   is_global_block: false
  }
 };
}

function read_block(parser) {
 const start = parser.index;

 parser.eat('{', true);

 /** @type {Array<AST.CSS.Declaration | AST.CSS.Rule | AST.CSS.Atrule>} */
 const children = [];
```

```
  while (parser.index < parser.template.length) {
   allow_comment_or_whitespace(parser);

   if (parser.match('}')) {
    break;
   } else {
    children.push(read_block_item(parser));
   }
  }

  parser.eat('}', true);

  return {
   type: 'Block',
   start,
   end: parser.index,
   children
  };
 }

 function read_block_item(parser) {
  if (parser.match('@')) {
   return read_at_rule(parser);
  }

  // read ahead to understand whether we're dealing with a declaration or a nested rule.
  // this involves some duplicated work, but avoids a try-catch that would disguise errors
  const start = parser.index;
  read_value(parser);
  const char = parser.template[parser.index];
  parser.index = start;

  return char === '{' ? read_rule(parser) : read_declaration(parser);
 }

 function read_declaration(parser) {
  const start = parser.index;

  const property = parser.read_until(REGEX_WHITESPACE_OR_COLON);
  parser.allow_whitespace();
  parser.eat(':');
  let index = parser.index;
  parser.allow_whitespace();

  const value = read_value(parser);

  if (!value && !property.startsWith('--')) {
   e.css_empty_declaration({ start, end: index });
  }

  const end = parser.index;

  if (!parser.match('}')) {
   parser.eat(';', true);
  }

  return {
   type: 'Declaration',
   start,
   end,
```

```
   property,
   value
  };
 }

 function read_value(parser) {
  let value = '';
  let escaped = false;
  let in_url = false;

  /** @type {null | '"' | "'"} */
  let quote_mark = null;

  while (parser.index < parser.template.length) {
   const char = parser.template[parser.index];

   if (escaped) {
    value += '\\' + char;
    escaped = false;
   } else if (char === '\\') {
    escaped = true;
   } else if (char === quote_mark) {
    quote_mark = null;
   } else if (char === ')') {
    in_url = false;
   } else if (quote_mark === null && (char === '"' || char === "'")) {
    quote_mark = char;
   } else if (char === '(' && value.slice(-3) === 'url') {
    in_url = true;
   } else if ((char === ';' || char === '{' || char === '}') && !in_url && !quote_mark) {
    return value.trim();
   }

   value += char;

   parser.index++;
  }

  throw new Error('Unexpected end of input');
 }

 function read_selector_list(parser, inside_pseudo_class = false) {
  /** @type {AST.CSS.ComplexSelector[]} */
  const children = [];

  allow_comment_or_whitespace(parser);

  const start = parser.index;

  while (parser.index < parser.template.length) {
   children.push(read_selector(parser, inside_pseudo_class));

   const end = parser.index;

   allow_comment_or_whitespace(parser);

   if (inside_pseudo_class ? parser.match(')') : parser.match('{')) {
    return {
     type: 'SelectorList',
     start,
```

```
    end,
    children
   };
  } else {
   parser.eat(',', true);
   allow_comment_or_whitespace(parser);
  }
 }

 throw new Error('Unexpected end of input');
}

function read_combinator(parser) {
 const start = parser.index;
 parser.allow_whitespace();

 const index = parser.index;
 const name = parser.read(REGEX_COMBINATOR);

 if (name) {
  const end = parser.index;
  parser.allow_whitespace();

  return {
   type: 'Combinator',
   name,
   start: index,
   end
  };
 }

 if (parser.index !== start) {
  return {
   type: 'Combinator',
   name: ' ',
   start,
   end: parser.index
  };
 }

 return null;
}

function read_selector(parser, inside_pseudo_class = false) {
 const list_start = parser.index;

 /** @type {AST.CSS.RelativeSelector[]} */
 const children = [];

 /**
     * @param {AST.CSS.Combinator | null} combinator
     * @param {number} start
     * @returns {AST.CSS.RelativeSelector}
     */
 function create_selector(combinator, start) {
  return {
   type: 'RelativeSelector',
   combinator,
   selectors: [],
   start,
```

```
    end: -1,
    metadata: {
     is_global: false,
     is_global_like: false,
     scoped: false
    }
   };
  }

  /** @type {AST.CSS.RelativeSelector} */
  let relative_selector = create_selector(null, parser.index);

  while (parser.index < parser.template.length) {
   let start = parser.index;

   if (parser.eat('&')) {
    relative_selector.selectors.push({
     type: 'NestingSelector',
     name: '&',
     start,
     end: parser.index
    });
   } else if (parser.eat('*')) {
    let name = '*';

    if (parser.eat('|')) {
     // * is the namespace (which we ignore)
     name = read_identifier(parser);
    }

    relative_selector.selectors.push({
     type: 'TypeSelector',
     name,
     start,
     end: parser.index
    });
   } else if (parser.eat('#')) {
    relative_selector.selectors.push({
     type: 'IdSelector',
     name: read_identifier(parser),
     start,
     end: parser.index
    });
   } else if (parser.eat('.')) {
    relative_selector.selectors.push({
     type: 'ClassSelector',
     name: read_identifier(parser),
     start,
     end: parser.index
    });
   } else if (parser.eat('::')) {
    relative_selector.selectors.push({
     type: 'PseudoElementSelector',
     name: read_identifier(parser),
     start,
     end: parser.index
    });
    // We read the inner selectors of a pseudo element to ensure it parses correctly,
    // but we don't do anything with the result.
    if (parser.eat('(')) {
```

```
     read_selector_list(parser, true);
     parser.eat(')', true);
    }
   } else if (parser.eat(':')) {
    const name = read_identifier(parser);

    /** @type {null | AST.CSS.SelectorList} */
    let args = null;

    if (parser.eat('(')) {
     args = read_selector_list(parser, true);
     parser.eat(')', true);
    }

    relative_selector.selectors.push({
     type: 'PseudoClassSelector',
     name,
     args,
     start,
     end: parser.index
    });
   } else if (parser.eat('[')) {
    parser.allow_whitespace();
    const name = read_identifier(parser);
    parser.allow_whitespace();

    /** @type {string | null} */
    let value = null;

    const matcher = parser.read(REGEX_MATCHER);

    if (matcher) {
     parser.allow_whitespace();
     value = read_attribute_value(parser);
    }

    parser.allow_whitespace();

    const flags = parser.read(REGEX_ATTRIBUTE_FLAGS);

    parser.allow_whitespace();
    parser.eat(']', true);

    relative_selector.selectors.push({
     type: 'AttributeSelector',
     start,
     end: parser.index,
     name,
     matcher,
     value,
     flags
    });
   } else if (inside_pseudo_class && parser.match_regex(REGEX_NTH_OF)) {
    // nth of matcher must come before combinator matcher to prevent collision else the '+' in '+2n-
1' would be parsed as a combinator

    relative_selector.selectors.push({
     type: 'Nth',
     value: /**@type {string} */ (parser.read(REGEX_NTH_OF)),
     start,
```

```
     end: parser.index
    });
   } else if (parser.match_regex(REGEX_PERCENTAGE)) {
    relative_selector.selectors.push({
     type: 'Percentage',
     value: /** @type {string} */ (parser.read(REGEX_PERCENTAGE)),
     start,
     end: parser.index
    });
   } else if (!parser.match_regex(REGEX_COMBINATOR)) {
    let name = read_identifier(parser);

    if (parser.eat('|')) {
     // we ignore the namespace when trying to find matching element classes
     name = read_identifier(parser);
    }

    relative_selector.selectors.push({
     type: 'TypeSelector',
     name,
     start,
     end: parser.index
    });
   }

   const index = parser.index;
   allow_comment_or_whitespace(parser);

   if (parser.match(',') || (inside_pseudo_class ? parser.match(')') : parser.match('{'))) {
    // rewind, so we know whether to continue building the selector list
    parser.index = index;

    relative_selector.end = index;
    children.push(relative_selector);

    return {
     type: 'ComplexSelector',
     start: list_start,
     end: index,
     children,
     metadata: {
      rule: null,
      used: false
     }
    };
   }

   parser.index = index;
   const combinator = read_combinator(parser);

   if (combinator) {
    if (relative_selector.selectors.length > 0) {
     relative_selector.end = index;
     children.push(relative_selector);
    }

    // ...and start a new one
    relative_selector = create_selector(combinator, combinator.start);

    parser.allow_whitespace();
```

```
   if (parser.match(',') || (inside_pseudo_class ? parser.match(')') : parser.match('{'))) {
    e.css_selector_invalid(parser.index);
   }
  }
 }

 throw new Error('Unexpected end of input');
}

function read_identifier(parser) {
 const start = parser.index;

 let identifier = '';

 if (parser.match_regex(REGEX_LEADING_HYPHEN_OR_DIGIT)) {
  throw new Error('Unexpected CSS identifier');
 }

 let escaped = false;

 while (parser.index < parser.template.length) {
  const char = parser.template[parser.index];
  if (escaped) {
   identifier += '\\' + char;
   escaped = false;
  } else if (char === '\\') {
   escaped = true;
  } else if (
   /** @type {number} */ (char.codePointAt(0)) >= 160 ||
   REGEX_VALID_IDENTIFIER_CHAR.test(char)
  ) {
   identifier += char;
  } else {
   break;
  }

  parser.index++;
 }

 if (identifier === '') {
  throw new Error('Expected identifier');
 }

 return identifier;
}
```

src/compiler/phases/2-analyze/index.js

```
import * as b from '../../../utils/builders.js';
import { walk } from 'zimmerframe';
import { create_scopes, ScopeRoot } from '../../scope.js';
import {
 get_delegated_event,
 is_event_attribute,
 is_inside_component,
 is_svelte_import,
 is_tracked_name
```

```
} from '../../utils.js';
import { extract_paths } from '../../../utils/ast.js';
import is_reference from 'is-reference';
import { prune_css } from './prune.js';
import { error } from '../../errors.js';

function visit_function(node, context) {
 node.metadata = {
  hoisted: false,
  hoisted_params: [],
  scope: context.state.scope,
  tracked: false
 };

 if (node.params.length > 0) {
  for (let i = 0; i < node.params.length; i += 1) {
   const param = node.params[i];
   if (param.type === 'ObjectPattern') {
    const paths = extract_paths(param);

    for (const path of paths) {
     const name = path.node.name;
     const binding = context.state.scope.get(name);

     if (binding !== null && is_tracked_name(name)) {
      const id = context.state.scope.generate('arg');
      node.params[i] = b.id(id);
      binding.kind = 'prop';

      binding.transform = {
       read: (_) => b.call('$.get_property', b.id(id), b.literal(name))
      };
     }
    }
   }
  }
 }

 context.next({
  ...context.state,
  function_depth: context.state.function_depth + 1,
  expression: null
 });
}

function mark_as_tracked(path) {
 for (let i = 0; i < path.length; i += 1) {
  const node = path[i];

  if (node.type === 'Component') {
   break;
  }
  if (
   node.type === 'FunctionExpression' ||
   node.type === 'ArrowFunctionExpression' ||
   node.type === 'FunctionDeclaration'
  ) {
   node.metadata.tracked = true;
   break;
  }
```

```
  }
 }

 const visitors = {
  _(node, { state, next }) {
   const scope = state.scopes.get(node);
   next(scope !== undefined && scope !== state.scope ? { ...state, scope } : state);
  },

  Identifier(node, context) {
   const binding = context.state.scope.get(node.name);
   const parent = context.path.at(-1);

   if (
    is_reference(node, /** @type {Node} */ (parent)) &&
    context.state.metadata?.tracking === false &&
    is_tracked_name(node.name) &&
    binding?.node !== node
   ) {
    context.state.metadata.tracking = true;
   }

   context.next();
  },

  MemberExpression(node, context) {
   const parent = context.path.at(-1);

   if (
    context.state.metadata?.tracking === false &&
    node.property.type === 'Identifier' &&
    !node.computed &&
    is_tracked_name(node.property.name) &&
    parent.type !== 'AssignmentExpression'
   ) {
    context.state.metadata.tracking = true;
   }
   context.next();
  },

  CallExpression(node, context) {
   if (context.state.metadata?.tracking === false) {
    context.state.metadata.tracking = true;
   }

   context.next();
  },

  ObjectExpression(node, context) {
   for (const property of node.properties) {
    if (
     property.type === 'Property' &&
     !property.computed &&
     property.key.type === 'Identifier' &&
     property.kind === 'init' &&
     is_tracked_name(property.key.name)
    ) {
     mark_as_tracked(context.path);
    }
   }
```

```
 context.next();
},

ArrayExpression(node, context) {
 for (const element of node.elements) {
  if (element !== null && element.type === 'Identifier' && is_tracked_name(element.name)) {
   mark_as_tracked(context.path);
  }
 }

 context.next();
},

VariableDeclaration(node, context) {
 const { state, visit, path } = context;

 for (const declarator of node.declarations) {
  const metadata = { tracking: false, await: false };
  const parent = path.at(-1);
  const init_is_untracked =
   declarator.init !== null &&
   declarator.init.type === 'CallExpression' &&
   is_svelte_import(declarator.init.callee, context) &&
   declarator.init.callee.type === 'Identifier' &&
   (declarator.init.callee.name === 'untrack' || declarator.init.callee.name === 'deferred');

  if (declarator.id.type === 'Identifier') {
   const binding = state.scope.get(declarator.id.name);

   if (
    binding !== null &&
    is_tracked_name(declarator.id.name) &&
    parent?.type !== 'ForOfStatement'
   ) {
    binding.kind = 'tracked';

    mark_as_tracked(path);

    visit(declarator, { ...state, metadata });

    if (init_is_untracked && metadata.tracking) {
     metadata.tracking = false;
    }

    binding.transform = {
     read: (node) => {
      return metadata.tracking && !metadata.await
       ? b.call('$.get_computed', node)
       : b.call('$.get_tracked', node);
     },
     assign: (node, value) => {
      return b.call('$.set', node, value, b.id('__block'));
     },
     update: (node) => {
      return b.call(
       node.prefix ? '$.update_pre' : '$.update',
       node.argument,
       b.id('__block'),
       node.operator === '--' && b.literal(-1)
```

```
        );
       }
      };
     } else {
      visit(declarator, state);
     }
    } else {
     const paths = extract_paths(declarator.id);
     const has_tracked = paths.some(
      (path) => path.node.type === 'Identifier' && is_tracked_name(path.node.name)
     );

     if (has_tracked) {
      const tmp = state.scope.generate('tmp');
      declarator.transformed = b.id(tmp);

      if (declarator.init !== null) {
       visit(declarator.init, { ...state, metadata });
      }

      if (init_is_untracked && metadata.tracking) {
       metadata.tracking = false;
      }

      for (const path of paths) {
       const binding = state.scope.get(path.node.name);

       binding.transform = {
        read: (node) => {
         const value = path.expression?.(b.id(tmp));

         if (metadata.tracking && metadata.await) {
          // TODO
          debugger;
         } else if (metadata.tracking && !metadata.await) {
          if (is_tracked_name(path.node.name) && value.type === 'MemberExpression') {
           return b.call(
            '$.get_property',
            b.call('$.get_computed', value.object),
            value.property.type === 'Identifier'
             ? b.literal(value.property.name)
             : value.property
           );
          }

          const key =
           value.property.type === 'Identifier'
            ? b.key(value.property.name)
            : value.property;

          return b.member(
           b.call('$.get_computed', value.object),
           key,
           key.type === 'Literal'
          );
         }

         if (is_tracked_name(path.node.name) && value.type === 'MemberExpression') {
          return b.call(
           '$.get_property',
```

```
          value.object,
          value.property.type === 'Identifier'
           ? b.literal(value.property.name)
           : value.property
         );
        }

        return value;
       }
      };
     }
    } else {
     visit(declarator, state);
    }
   }

   declarator.metadata = metadata;
  }
 },

 ArrowFunctionExpression(node, context) {
  visit_function(node, context);
 },
 FunctionExpression(node, context) {
  visit_function(node, context);
 },
 FunctionDeclaration(node, context) {
  visit_function(node, context);
 },

 Component(node, context) {
  context.state.component = node;

  if (node.params.length > 0) {
   const props = node.params[0];

   if (props.type === 'ObjectPattern') {
    const paths = extract_paths(props);

    for (const path of paths) {
     const name = path.node.name;
     const binding = context.state.scope.get(name);

     if (binding !== null && is_tracked_name(name)) {
      binding.kind = 'prop';

      binding.transform = {
       read: (_) => b.call('$.get_property', b.id('__props'), b.literal(name))
      };
     }
    }
   }
  }
  const elements = [];

  context.next({ ...context.state, elements });

  const css = node.css;

  if (css !== null) {
```

```
    for (const node of elements) {
     prune_css(css, node);
    }
   }
  },

  ForStatement(node, context) {
   if (is_inside_component(context)) {
    error(
     'For loops are not supported in components. Use for...of instead.',
     context.state.analysis.module.filename,
     node
    );
   }

   context.next();
  },

  ForInStatement(node, context) {
   if (is_inside_component(context)) {
    error(
     'For...in loops are not supported in components. Use for...of instead.',
     context.state.analysis.module.filename,
     node
    );
   }

   context.next();
  },

  JSXElement(_, context) {
   {
    error(
     'Elements cannot be used as generic expressions, only as statements within a component',
     context.state.analysis.module.filename,
     node
    );
   }
  },

  Element(node, { state, visit }) {
   const type = node.id.name;
   const is_dom_element = type[0].toLowerCase() === type[0];
   const attribute_names = new Set();

   if (is_dom_element) {
    if (state.elements) {
     state.elements.push(node);
    }

    for (const attr of node.attributes) {
     if (attr.type === 'Attribute') {
      if (attr.name.type === 'Identifier') {
       attribute_names.add(attr.name);

       if (is_event_attribute(attr.name.name)) {
        const event_name = attr.name.name.slice(2).toLowerCase();
        const handler = visit(attr.value, state);
        const delegated_event = get_delegated_event(event_name, handler, state);
```

```
        if (delegated_event !== null) {
         if (delegated_event.hoisted) {
          delegated_event.function.metadata.hoisted = true;
          delegated_event.hoisted = true;
         }

         if (attr.metadata === undefined) {
          attr.metadata = {};
         }

         attr.metadata.delegated = delegated_event;
        }
       }
      }
     }
    }
   } else {
    for (const attr of node.attributes) {
     if (attr.type === 'Attribute') {
      if (attr.name.type === 'Identifier') {
       attribute_names.add(attr.name);
      }
     }
    }

    let implicit_children = false;
    let explicit_children = false;

    for (const child of node.children) {
     if (child.type === 'Fragment') {
      if (child.id.name === '$children') {
       explicit_children = true;
       if (implicit_children) {
        error(
         'Cannot have both implicit and explicit children',
         context.state.analysis.module.filename,
         node
        );
       }
      }
     } else if (child.type !== 'EmptyStatement') {
      implicit_children = true;
      if (explicit_children) {
       error(
        'Cannot have both implicit and explicit children',
        context.state.analysis.module.filename,
        node
       );
      }
     }
    }
   }

   // Validation
   for (const attribute of attribute_names) {
    const name = attribute.name;
    if (name === 'children') {
     if (is_dom_element) {
      error(
       'Cannot have a `children` prop on an element',
```

```
      state.analysis.module.filename,
      attribute
     );
    } else {
     error(
      'Cannot have a `children` prop on a component, did you mean `$children`?',
      state.analysis.module.filename,
      attribute
     );
    }
   } else if (name === 'ref') {
    if (is_dom_element) {
     error(
      'Cannot have a `ref` prop on an element, did you mean `$ref`?',
      state.analysis.module.filename,
      attribute
     );
    } else {
     error(
      'Cannot have a `ref` prop on a component, did you mean `$ref`?',
      state.analysis.module.filename,
      attribute
     );
    }
   }

   if (is_tracked_name(name)) {
    attribute_names.forEach((n) => {
     if (n.name.slice(1) === name) {
      error(
       `Cannot have both ${name} and ${name.slice(1)} on the same element`,
       state.analysis.module.filename,
       n
      );
     }
    });
   }
  }

  return {
   ...node,
   children: node.children.map((child) => visit(child))
  };
 },

 AwaitExpression(node, context) {
  if (is_inside_component(context)) {
   if (context.state.metadata?.await === false) {
    context.state.metadata.await = true;
   }
  }

  context.next();
 }
};

export function analyze(ast, filename) {
 const scope_root = new ScopeRoot();

 const { scope, scopes } = create_scopes(ast, scope_root, null);
```

```
 const analysis = {
  module: { ast, scope, scopes, filename },
  ast,
  scope,
  scopes
 };

 walk(
  ast,
  {
   scope,
   scopes,
   analysis
  },
  visitors
 );

 return analysis;
}
```

src/compiler/phases/2-analyze/prune.js

```
import { walk } from 'zimmerframe';

const seen = new Set();
const regex_backslash_and_following_character = /\\(.)/g;

function get_relative_selectors(node) {
 const selectors = truncate(node);

 if (node.metadata.rule?.metadata.parent_rule && selectors.length > 0) {
  let has_explicit_nesting_selector = false;

  // nesting could be inside pseudo classes like :is, :has or :where
  for (let selector of selectors) {
   walk(selector, null, {
    // @ts-ignore
    NestingSelector() {
     has_explicit_nesting_selector = true;
    }
   });

   // if we found one we can break from the others
   if (has_explicit_nesting_selector) break;
  }

  if (!has_explicit_nesting_selector) {
   if (selectors[0].combinator === null) {
    selectors[0] = {
     ...selectors[0],
     combinator: descendant_combinator
    };
   }

   selectors.unshift(nesting_selector);
  }
 }
```

```
 return selectors;
}

function truncate(node) {
 const i = node.children.findLastIndex(({ metadata, selectors }) => {
  const first = selectors[0];
  return (
   // not after a :global selector
   !metadata.is_global_like &&
   !(first.type === 'PseudoClassSelector' && first.name === 'global' && first.args === null) &&
   // not a :global(...) without a :has/is/where(...) modifier that is scoped
   !metadata.is_global
  );
 });

 return node.children.slice(0, i + 1).map((child) => {
  // In case of `:root.y:has(...)`, `y` is unscoped, but everything in `:has(...)` should be scoped
(if not global).
  // To properly accomplish that, we gotta filter out all selector types except `:has`.
  const root = child.selectors.find((s) => s.type === 'PseudoClassSelector' && s.name === 'root');
  if (!root || child.metadata.is_global_like) return child;

  return {
   ...child,
   selectors: child.selectors.filter((s) => s.type === 'PseudoClassSelector' && s.name === 'has')
  };
 });
}

function apply_selector(relative_selectors, rule, element) {
 const parent_selectors = relative_selectors.slice();
 const relative_selector = parent_selectors.pop();

 const matched =
  !!relative_selector &&
  relative_selector_might_apply_to_node(relative_selector, rule, element) &&
  apply_combinator(relative_selector, parent_selectors, rule, element);

 if (matched) {
  if (!is_outer_global(relative_selector)) {
   relative_selector.metadata.scoped = true;
  }

  element.metadata.scoped = true;
 }

 return matched;
}

function apply_combinator(relative_selector, parent_selectors, rule, node) {
 if (!relative_selector.combinator) return true;

 const name = relative_selector.combinator.name;

 switch (name) {
  case ' ':
  case '>': {
   let parent_matched = false;
```

```
    const path = node.metadata.path;
    let i = path.length;

    while (i--) {
     const parent = path[i];

     if (parent.type === 'Element') {
      if (apply_selector(parent_selectors, rule, parent)) {
       parent_matched = true;
      }

      if (name === '>') return parent_matched;
     }
    }

    return parent_matched || parent_selectors.every((selector) => is_global(selector, rule));
   }

   case '+':
   case '~': {
    const siblings = get_possible_element_siblings(node, name === '+');

    let sibling_matched = false;

    for (const possible_sibling of siblings.keys()) {
     if (possible_sibling.type === 'RenderTag' || possible_sibling.type === 'SlotElement') {
      // `{@render foo()}<p>foo</p>` with `:global(.x) + p` is a match
      if (parent_selectors.length === 1 && parent_selectors[0].metadata.is_global) {
       sibling_matched = true;
      }
     } else if (apply_selector(parent_selectors, rule, possible_sibling)) {
      sibling_matched = true;
     }
    }

    return (
     sibling_matched ||
     (get_element_parent(node) === null &&
      parent_selectors.every((selector) => is_global(selector, rule)))
    );
   }

   default:
    // TODO other combinators
    return true;
  }
 }

function get_element_parent(node) {
 let path = node.metadata.path;
 let i = path.length;

 while (i--) {
  const parent = path[i];

  if (parent.type === 'RegularElement' || parent.type === 'SvelteElement') {
   return parent;
  }
 }
```

```
 return null;
}

function is_global(selector, rule) {
 if (selector.metadata.is_global || selector.metadata.is_global_like) {
  return true;
 }

 for (const s of selector.selectors) {
  /** @type {Compiler.AST.CSS.SelectorList | null} */
  let selector_list = null;
  let owner = rule;

  if (s.type === 'PseudoClassSelector') {
   if ((s.name === 'is' || s.name === 'where') && s.args) {
    selector_list = s.args;
   }
  }

  if (s.type === 'NestingSelector') {
   owner = /** @type {Compiler.AST.CSS.Rule} */ (rule.metadata.parent_rule);
   selector_list = owner.prelude;
  }

  const has_global_selectors = selector_list?.children.some((complex_selector) => {
   return complex_selector.children.every((relative_selector) =>
    is_global(relative_selector, owner)
   );
  });

  if (!has_global_selectors) {
   return false;
  }
 }

 return true;
}

function is_text_attribute(attribute) {
 return attribute.value.type === 'Literal';
}

function test_attribute(operator, expected_value, case_insensitive, value) {
 if (case_insensitive) {
  expected_value = expected_value.toLowerCase();
  value = value.toLowerCase();
 }
 switch (operator) {
  case '=':
   return value === expected_value;
  case '~=':
   return value.split(/\s/).includes(expected_value);
  case '|=':
   return `${value}-`.startsWith(`${expected_value}-`);
  case '^=':
   return value.startsWith(expected_value);
  case '$=':
   return value.endsWith(expected_value);
  case '*=':
   return value.includes(expected_value);
```

```
   default:
    throw new Error("this shouldn't happen");
  }
 }

 function attribute_matches(node, name, expected_value, operator, case_insensitive) {
  for (const attribute of node.attributes) {
   if (attribute.type === 'SpreadAttribute') return true;

   if (attribute.type !== 'Attribute') continue;
   if (attribute.name.name.toLowerCase() !== name.toLowerCase()) continue;

   if (expected_value === null) return true;

   if (is_text_attribute(attribute)) {
    return test_attribute(operator, expected_value, case_insensitive, attribute.value.value);
   } else {
    return true;
   }
  }

  return false;
 }

 function is_outer_global(relative_selector) {
  const first = relative_selector.selectors[0];

  return (
   first.type === 'PseudoClassSelector' &&
   first.name === 'global' &&
   (first.args === null ||
    // Only these two selector types can keep the whole selector global, because e.g.
    // :global(button).x means that the selector is still scoped because of the .x
    relative_selector.selectors.every(
     (selector) =>
      selector.type === 'PseudoClassSelector' || selector.type === 'PseudoElementSelector'
    ))
  );
 }

 function relative_selector_might_apply_to_node(relative_selector, rule, element) {
  // Sort :has(...) selectors in one bucket and everything else into another
  const has_selectors = [];
  const other_selectors = [];

  for (const selector of relative_selector.selectors) {
   if (selector.type === 'PseudoClassSelector' && selector.name === 'has' && selector.args) {
    has_selectors.push(selector);
   } else {
    other_selectors.push(selector);
   }
  }

  // If we're called recursively from a :has(...) selector, we're on the way of checking if the other
 selectors match.
  // In that case ignore this check (because we just came from this) to avoid an infinite loop.
  if (has_selectors.length > 0) {
   /** @type {Array<Compiler.AST.RegularElement | Compiler.AST.SvelteElement>} */
   const child_elements = [];
   /** @type {Array<Compiler.AST.RegularElement | Compiler.AST.SvelteElement>} */
```

```
  const descendant_elements = [];
  /** @type {Array<Compiler.AST.RegularElement | Compiler.AST.SvelteElement>} */
  let sibling_elements; // do them lazy because it's rarely used and expensive to calculate

  // If this is a :has inside a global selector, we gotta include the element itself, too,
  // because the global selector might be for an element that's outside the component,
  // e.g. :root:has(.scoped), :global(.foo):has(.scoped), or :root { &:has(.scoped) {} }
  const rules = get_parent_rules(rule);
  const include_self =
   rules.some((r) => r.prelude.children.some((c) => c.children.some((s) => is_global(s, r)))) ||
   rules[rules.length - 1].prelude.children.some((c) =>
    c.children.some((r) =>
     r.selectors.some(
      (s) =>
       s.type === 'PseudoClassSelector' &&
       (s.name === 'root' || (s.name === 'global' && s.args))
     )
    )
   );
  if (include_self) {
   child_elements.push(element);
   descendant_elements.push(element);
  }

  /**
         * @param {Compiler.AST.SvelteNode} node
         * @param {{ is_child: boolean }} state
         */
  function walk_children(node, state) {
   walk(node, state, {
    _(node, context) {
     if (node.type === 'Element') {
      descendant_elements.push(node);

      if (context.state.is_child) {
       child_elements.push(node);
       context.state.is_child = false;
       context.next();
       context.state.is_child = true;
      } else {
       context.next();
      }
     } else {
      context.next();
     }
    }
   });
  }

  walk_children(element.fragment, { is_child: true });

  // :has(...) is special in that it means "look downwards in the CSS tree". Since our matching algo
rithm goes
  // upwards and back-to-front, we need to first check the selectors inside :has(...), then check th
e rest of the
  // selector in a way that is similar to ancestor matching. In a sense, we're treating `.x:has(.y)`
as `.x .y`.
  for (const has_selector of has_selectors) {
   const complex_selectors = /** @type {Compiler.AST.CSS.SelectorList} */ (has_selector.args)
    .children;
```

```
    let matched = false;

    for (const complex_selector of complex_selectors) {
     const selectors = truncate(complex_selector);
     const left_most_combinator = selectors[0]?.combinator ?? descendant_combinator;
     // In .x:has(> y), we want to search for y, ignoring the left-most combinator
     // (else it would try to walk further up and fail because there are no selectors left)
     if (selectors.length > 0) {
      selectors[0] = {
       ...selectors[0],
       combinator: null
      };
     }

     const descendants =
      left_most_combinator.name === '+' || left_most_combinator.name === '~'
       ? (sibling_elements ??= get_following_sibling_elements(element, include_self))
       : left_most_combinator.name === '>'
       ? child_elements
       : descendant_elements;

     let selector_matched = false;

     // Iterate over all descendant elements and check if the selector inside :has matches
     for (const element of descendants) {
      if (
       selectors.length === 0 /* is :global(...) */ ||
       (element.metadata.scoped && selector_matched) ||
       apply_selector(selectors, rule, element)
      ) {
       complex_selector.metadata.used = true;
       selector_matched = matched = true;
      }
     }
    }

    if (!matched) {
     return false;
    }
   }
  }

  for (const selector of other_selectors) {
   if (selector.type === 'Percentage' || selector.type === 'Nth') continue;

   const name = selector.name.replace(regex_backslash_and_following_character, '$1');

   switch (selector.type) {
    case 'PseudoClassSelector': {
     if (name === 'host' || name === 'root') return false;

     if (
      name === 'global' &&
      selector.args !== null &&
      relative_selector.selectors.length === 1
     ) {
      const args = selector.args;
      const complex_selector = args.children[0];
      return apply_selector(complex_selector.children, rule, element);
     }
```

```
    // We came across a :global, everything beyond it is global and therefore a potential match
    if (name === 'global' && selector.args === null) return true;

    // :not(...) contents should stay unscoped. Scoping them would achieve the opposite of what we w
ant,
    // because they are then _more_ likely to bleed out of the component. The exception is complex s
electors
    // with descendants, in which case we scope them all.
    if (name === 'not' && selector.args) {
     for (const complex_selector of selector.args.children) {
      walk(complex_selector, null, {
       ComplexSelector(node, context) {
        node.metadata.used = true;
        context.next();
       }
      });
      const relative = truncate(complex_selector);

      if (complex_selector.children.length > 1) {
       // foo:not(bar foo) means that bar is an ancestor of foo (side note: ending with foo is the o
nly way the selector make sense).
       // We can't fully check if that actually matches with our current algorithm, so we just assum
e it does.
       // The result may not match a real element, so the only drawback is the missing prune.
       for (const selector of relative) {
        selector.metadata.scoped = true;
       }

       /** @type {Compiler.AST.RegularElement | Compiler.AST.SvelteElement | null} */
       let el = element;
       while (el) {
        el.metadata.scoped = true;
        el = get_element_parent(el);
       }
      }
     }

     break;
    }

    if ((name === 'is' || name === 'where') && selector.args) {
     let matched = false;

     for (const complex_selector of selector.args.children) {
      const relative = truncate(complex_selector);
      const is_global = relative.length === 0;

      if (is_global) {
       complex_selector.metadata.used = true;
       matched = true;
      } else if (apply_selector(relative, rule, element)) {
       complex_selector.metadata.used = true;
       matched = true;
      } else if (complex_selector.children.length > 1 && (name == 'is' || name == 'where')) {
       // foo :is(bar baz) can also mean that bar is an ancestor of foo, and baz a descendant.
       // We can't fully check if that actually matches with our current algorithm, so we just assum
e it does.
       // The result may not match a real element, so the only drawback is the missing prune.
       complex_selector.metadata.used = true;
```

```
        matched = true;
        for (const selector of relative) {
         selector.metadata.scoped = true;
        }
       }
      }

      if (!matched) {
       return false;
      }
     }

     break;
    }

    case 'PseudoElementSelector': {
     break;
    }

    case 'AttributeSelector': {
     const whitelisted = whitelist_attribute_selector.get(element.id.name.toLowerCase());
     if (
      !whitelisted?.includes(selector.id.name.toLowerCase()) &&
      !attribute_matches(
       element,
       selector.name,
       selector.value && unquote(selector.value),
       selector.matcher,
       selector.flags?.includes('i') ?? false
      )
     ) {
      return false;
     }
     break;
    }

    case 'ClassSelector': {
     if (
      !attribute_matches(element, 'class', name, '~=', false) &&
      !element.attributes.some(
       (attribute) => attribute.type === 'ClassDirective' && attribute.name === name
      )
     ) {
      return false;
     }

     break;
    }

    case 'IdSelector': {
     if (!attribute_matches(element, 'id', name, '=', false)) {
      return false;
     }

     break;
    }

    case 'TypeSelector': {
     if (
      element.id.name.toLowerCase() !== name.toLowerCase() &&
```

```
      name !== '*' &&
      element.id.name[0].toLowerCase() === element.id.name[0]
     ) {
      return false;
     }

     break;
    }

    case 'NestingSelector': {
     let matched = false;

     const parent = /** @type {Compiler.AST.CSS.Rule} */ (rule.metadata.parent_rule);

     for (const complex_selector of parent.prelude.children) {
      if (
       apply_selector(get_relative_selectors(complex_selector), parent, element) ||
       complex_selector.children.every((s) => is_global(s, parent))
      ) {
       complex_selector.metadata.used = true;
       matched = true;
      }
     }

     if (!matched) {
      return false;
     }

     break;
    }
   }
  }

  // possible match
  return true;
 }

 export function prune_css(css, element) {
  walk(css, null, {
   Rule(node, context) {
    if (node.metadata.is_global_block) {
     context.visit(node.prelude);
    } else {
     context.next();
    }
   },
   ComplexSelector(node) {
    const selectors = get_relative_selectors(node);

    seen.clear();

    if (
     apply_selector(
      selectors,
      /** @type {Compiler.AST.CSS.Rule} */ (node.metadata.rule),
      element
     )
    ) {
     node.metadata.used = true;
    }
```

```
    // note: we don't call context.next() here, we only recurse into
    // selectors that don't belong to rules (i.e. inside `:is(...)` etc)
    // when we encounter them below
  }
 });
}
```

src/compiler/phases/3-transform/index.js

```
import { walk } from 'zimmerframe';
import path from 'node:path';
import { print } from 'esrap';
import tsx from 'esrap/languages/tsx';
import * as b from '../../../utils/builders.js';
import { IS_CONTROLLED, TEMPLATE_FRAGMENT } from '../../../constants.js';
import { sanitize_template_string } from '../../../utils/sanitize_template_string.js';
import {
  build_hoisted_params,
  is_event_attribute,
  is_inside_component,
  is_tracked_name,
  is_passive_event,
  build_assignment,
  visit_assignment_expression,
  escape_html,
  is_boolean_attribute,
  is_dom_property,
  is_svelte_import,
  is_declared_within_component,
  is_inside_call_expression
} from '../../utils.js';
import is_reference from 'is-reference';
import { extract_paths, object } from '../../../utils/ast.js';
import { render_stylesheets } from './stylesheet.js';

function visit_function(node, context) {
  if (context.state.to_ts) {
    context.next(context.state);
    return;
  }
  const metadata = node.metadata;
  const state = context.state;

  if (metadata?.hoisted === true) {
    const params = build_hoisted_params(node, context);

    return /** @type {FunctionExpression} */ ({
      ...node,
      params,
      body: context.visit(node.body, state)
    });
  }

  let body = context.visit(node.body, state);

  if (metadata.tracked === true) {
    return /** @type {FunctionExpression} */ ({
```

```
      ...node,
      params: node.params.map((param) => context.visit(param, state)),
      body:
        body.type === 'BlockStatement'
          ? { ...body, body: [b.var('__block', b.call('$.scope')), ...body.body] }
          : body
    });
  }

  context.next(state);
}

function build_getter(node, context) {
  const state = context.state;

  for (let i = context.path.length - 1; i >= 0; i -= 1) {
    const binding = state.scope.get(node.name);

    // don't transform the declaration itself
    if (node !== binding?.node && binding?.transform) {
      return binding.transform.read(node);
    }
  }

  return node;
}

const visitors = {
  _: function set_scope(node, { next, state }) {
    const scope = state.scopes.get(node);

    if (scope && scope !== state.scope) {
      return next({ ...state, scope });
    } else {
      return next();
    }
  },

  Identifier(node, context) {
    const parent = /** @type {Node} */ (context.path.at(-1));

    if (is_reference(node, parent) && !context.state.to_ts) {
      const binding = context.state.scope.get(node.name);
      if (
        context.state.metadata?.tracking === false &&
        is_tracked_name(node.name) &&
        binding?.node !== node
      ) {
        context.state.metadata.tracking = true;
      }

      if (node.name === 'structuredClone' && binding === null) {
        return b.id('$.structured_clone');
      }

      return build_getter(node, context);
    }
  },

  ImportDeclaration(node, context) {
```

```
    if (!context.state.to_ts && node.importKind === 'type') {
      return b.empty;
    }
    return context.next();
  },

  CallExpression(node, context) {
    const callee = node.callee;
    const parent = context.path.at(-1);

    if (context.state.metadata?.tracking === false) {
      context.state.metadata.tracking = true;
    }

    if (
      context.state.to_ts ||
      (parent?.type === 'MemberExpression' && parent.property === node) ||
      is_inside_call_expression(context) ||
      !context.path.some((node) => node.type === 'Component') ||
      (is_svelte_import(callee, context) &&
        (callee.type !== 'Identifier' ||
          (callee.name !== 'array' && callee.name !== 'deferred'))) ||
      is_declared_within_component(callee, context)
    ) {
      return context.next();
    }

    return b.call(
      '$.with_scope',
      b.id('__block'),
      b.thunk({
        ...node,
        callee: context.visit(callee),
        arguments: node.arguments.map((arg) => context.visit(arg))
      })
    );
  },

  MemberExpression(node, context) {
    const parent = context.path.at(-1);

    if (parent.type !== 'AssignmentExpression') {
      const object = node.object;
      const property = node.property;
      const tracked_name =
        property.type === 'Identifier'
          ? is_tracked_name(property.name)
          : property.type === 'Literal' && is_tracked_name(property.value);

      // TODO should we enforce that the identifier is tracked too?
      if ((node.computed && property.type === 'Identifier') || tracked_name) {
        if (context.state.metadata?.tracking === false) {
          context.state.metadata.tracking = true;
        }

        if (tracked_name) {
          return b.call(
            '$.get_property',
            context.visit(object),
            property.type === 'Identifier' ? b.literal(property.name) : property,
```

```
            node.optional ? b.true : undefined
          );
        } else {
          return b.call(
            '$.get_property',
            context.visit(object),
            context.visit(property),
            node.optional ? b.true : undefined
          );
        }
      }

      if (object.type === 'Identifier' && object.name === 'Object') {
        const binding = context.state.scope.get(object.name);

        if (binding === null) {
          if (property.type === 'Identifier' && property.name === 'values') {
            return b.id('$.object_values');
          } else if (property.type === 'Identifier' && property.name === 'entries') {
            return b.id('$.object_entries');
          } else if (property.type === 'Identifier' && property.name === 'keys') {
            return b.id('$.object_keys');
          }
        }
      }
    }

    if (node.object.type === 'MemberExpression' && node.object.optional) {
      const metadata = { tracking: false, await: false };

      const object = context.visit(node.object, { ...context.state, metadata });

      if (metadata.tracking) {
        if (context.state.metadata?.tracking === false) {
          context.state.metadata.tracking = true;
        }

        return {
          ...node,
          optional: true,
          object,
          property: context.visit(node.property)
        };
      }
      if (metadata.await) {
        if (context.state.metadata?.await === false) {
          context.state.metadata.await = true;
        }
      }
    } else {
      context.next();
    }
  },

  SpreadElement(node, context) {
    const parent = context.path.at(-1);

    if (parent.type === 'ObjectExpression') {
      return b.spread(b.call('$.spread_object', context.visit(node.argument)));
    }
```

```
    context.next();
  },

  VariableDeclaration(node, context) {
    const declarations = [];

    for (const declarator of node.declarations) {
      const metadata = declarator.metadata;

      if (declarator.id.type === 'Identifier') {
        const binding = context.state.scope.get(declarator.id.name);

        if (!context.state.to_ts) {
          delete declarator.id.typeAnnotation;
        }

        if (binding !== null && binding.kind === 'tracked' && !context.state.to_ts) {
          let expression;

          if (metadata.tracking && metadata.await) {
            expression = b.call(
              b.await(
                b.call(
                  '$.resume_context',
                  b.call(
                    '$.async_computed',
                    b.thunk(context.visit(declarator.init), true),
                    b.id('__block')
                  )
                )
              )
            );
          } else if (metadata.tracking && !metadata.await) {
            expression = b.call(
              '$.computed',
              b.thunk(context.visit(declarator.init)),
              b.id('__block')
            );
          } else {
            expression = b.call(
              '$.tracked',
              declarator.init === null ? undefined : context.visit(declarator.init),
              b.id('__block')
            );
          }

          declarations.push(b.declarator(declarator.id, expression));
        } else {
          declarations.push(context.visit(declarator));
        }
      } else {
        const paths = extract_paths(declarator.id);
        const has_tracked = paths.some(
          (path) => path.node.type === 'Identifier' && is_tracked_name(path.node.name)
        );

        if (!context.state.to_ts) {
          delete declarator.id.typeAnnotation;
        }
```

```
        if (!has_tracked || context.state.to_ts) {
          declarations.push(context.visit(declarator));
          continue;
        }

        const transformed = declarator.transformed;
        let expression;

        if (metadata.tracking && metadata.await) {
          // TODO
          debugger;
        } else if (metadata.tracking && !metadata.await) {
          expression = b.call(
            '$.computed',
            b.thunk(context.visit(declarator.init)),
            b.id('__block')
          );
        } else {
          expression = context.visit(declarator.init);
        }

        declarations.push(b.declarator(transformed, expression));
      }
    }

    return { ...node, declarations };
  },

  FunctionDeclaration(node, context) {
    return visit_function(node, context);
  },
  ArrowFunctionExpression(node, context) {
    return visit_function(node, context);
  },
  FunctionExpression(node, context) {
    return visit_function(node, context);
  },

  Element(node, context) {
    const { state, visit } = context;

    const type = node.id.name;
    const is_dom_element = type[0].toLowerCase() === type[0];

    const handle_static_attr = (name, value) => {
      state.template.push(
        b.literal(
          ` ${name}${
            is_boolean_attribute(name) && value === true
              ? ''
              : `="${value === true ? '' : escape_html(value, true)}"`
          }`
        )
      );
    };

    if (is_dom_element) {
      let class_attribute = null;
```

```
      state.template.push(`<${type}`);

      for (const attr of node.attributes) {
        if (attr.type === 'Attribute') {
          if (attr.name.type === 'Identifier') {
            const name = attr.name.name;

            if (attr.value.type === 'Literal' && name !== 'class') {
              handle_static_attr(name, attr.value.value);
              continue;
            }

            if (name === 'class' || name === '$class') {
              class_attribute = attr;

              continue;
            }

            if (name === 'value' || name === '$value') {
              const id = state.flush_node();
              const expression = visit(attr.value, state);

              if (name === '$value') {
                state.update.push(b.stmt(b.call('$.set_value', id, expression)));
              } else {
                state.init.push(b.stmt(b.call('$.set_value', id, expression)));
              }

              continue;
            }

            if (name === 'checked' || name === '$checked') {
              const id = state.flush_node();
              const expression = visit(attr.value, state);

              if (name === '$checked') {
                state.update.push(b.stmt(b.call('$.set_checked', id, expression)));
              } else {
                state.init.push(b.stmt(b.call('$.set_checked', id, expression)));
              }
              continue;
            }

            if (name === 'selected' || name === '$selected') {
              const id = state.flush_node();
              const expression = visit(attr.value, state);

              if (name === '$selected') {
                state.update.push(b.stmt(b.call('$.set_selected', id, expression)));
              } else {
                state.init.push(b.stmt(b.call('$.set_selected', id, expression)));
              }
              continue;
            }

            if (name === '$ref') {
              const id = state.flush_node();
              const value = attr.value;

              state.init.push(b.stmt(b.call('$.set_ref', id, visit(value, state))));
```

```
            continue;
          }

          if (is_event_attribute(name)) {
            const event_name = name.slice(2).toLowerCase();
            let handler = visit(attr.value, state);
            let capture = false; // TODO

            if (attr.metadata?.delegated) {
              let delegated_assignment;

              if (!state.events.has(event_name)) {
                state.events.add(event_name);
              }

              // Hoist function if we can, otherwise we leave the function as is
              if (attr.metadata.delegated.hoisted) {
                if (attr.metadata.delegated.function === attr.value) {
                  const func_name = state.scope.root.unique('on_' + event_name);
                  state.hoisted.push(b.var(func_name, handler));
                  handler = func_name;
                }

                const hoisted_params = /** @type {Expression[]} */ (
                  attr.metadata.delegated.function.metadata.hoisted_params
                );

                const args = [handler, b.id('__block'), ...hoisted_params];
                delegated_assignment = b.array(args);
              } else if (
                handler.type === 'Identifier' &&
                is_declared_within_component(handler, context)
              ) {
                delegated_assignment = handler;
              } else {
                delegated_assignment = b.array([handler, b.id('__block')]);
              }
              const id = state.flush_node();

              state.init.push(
                b.stmt(b.assignment('=', b.member(id, '__' + event_name), delegated_assignment))
              );
            } else {
              const passive = is_passive_event(event_name);
              const id = state.flush_node();

              state.init.push(
                b.stmt(
                  b.call(
                    '$.event',
                    b.literal(event_name),
                    id,
                    handler,
                    capture && b.true,
                    passive === undefined ? undefined : b.literal(passive)
                  )
                )
              );
            }
```

```
            continue;
          }

          // All other attributes
          if (is_tracked_name(name)) {
            const attribute = name.slice(1);
            const id = state.flush_node();
            const expression = visit(attr.value, state);

            if (is_dom_property(attribute)) {
              state.update.push(b.stmt(b.assignment('=', b.member(id, attribute), expression)));
            } else {
              state.update.push(
                b.stmt(b.call('$.set_attribute', id, b.literal(attribute), expression))
              );
            }
          } else {
            const id = state.flush_node();
            const expression = visit(attr.value, state);

            if (is_dom_property(name)) {
              state.init.push(b.stmt(b.assignment('=', b.member(id, name), expression)));
            } else {
              state.init.push(b.stmt(b.call('$.set_attribute', id, b.literal(name), expression)));
            }
          }
        }
      }

      if (class_attribute !== null) {
        if (class_attribute.value.type === 'Literal') {
          let value = class_attribute.value.value;

          if (node.metadata.scoped && state.component.css) {
            value = `${state.component.css.hash} ${value}`;
          }

          handle_static_attr(class_attribute.name.name, value);
        } else {
          const id = state.flush_node();
          let expression = visit(class_attribute.value, state);

          if (node.metadata.scoped && state.component.css) {
            expression = b.binary('+', b.literal(state.component.css.hash + ' '), expression);
          }

          if (class_attribute.name.name === '$class') {
            state.update.push(b.stmt(b.call('$.set_class', id, expression)));
          } else {
            state.init.push(b.stmt(b.call('$.set_class', id, expression)));
          }
        }
      } else if (node.metadata.scoped && state.component.css) {
        const value = state.component.css.hash;

        handle_static_attr('class', value);
      }

      state.template.push('>');
```

```
      transform_children(node.children, { visit, state, root: false });

      state.template.push(`</${type}>`);
    } else {
      if (node.id.type !== 'Identifier') {
        throw new Error('TODO');
      }
      const id = state.flush_node();

      state.template.push('<!>');

      const tracked = [];
      let props = [];
      let children_prop = null;

      for (const attr of node.attributes) {
        if (attr.type === 'Attribute') {
          if (attr.name.type === 'Identifier' && is_tracked_name(attr.name.name)) {
            const metadata = { tracking: false, await: false };
            let property = visit(attr.value, { ...state, metadata });

            tracked.push(b.literal(attr.name.name));

            if (metadata.tracking) {
              const thunk = b.thunk(property);
              property = b.call('$.computed_property', thunk, b.id('__block'));

              if (attr.name.name === '$children') {
                children_prop = thunk;
              }
            }

            props.push(b.prop('init', attr.name, property));
          } else {
            props.push(b.prop('init', attr.name, visit(attr.value, state)));
          }
        } else {
          throw new Error('TODO');
        }
      }

      if (node.children.length > 0) {
        const component_scope = context.state.scopes.get(node);
        const children = b.arrow(
          [b.id('__anchor')],
          b.block(
            transform_body(node.children, {
              ...context,
              state: { ...context.state, scope: component_scope }
            })
          )
        );
        if (children_prop) {
          children_prop.body = b.logical('??', children_prop.body, children);
        } else {
          props.push(b.prop('init', b.id('$children'), children));
        }
      }
```

```
      if (tracked.length > 0) {
        state.init.push(
          b.stmt(
            b.call(
              node.id,
              id,
              b.call('$.tracked_object', b.object(props), b.array(tracked), b.id('__block')),
              b.id('$.active_block')
            )
          )
        );
      } else {
        state.init.push(b.stmt(b.call(node.id, id, b.object(props), b.id('$.active_block'))));
      }
    }
  },

  Fragment(node, context) {
    if (!context.state.to_ts) {
      if (!context.state.imports.has(`import * as $ from 'ripple/internal/client'`)) {
        context.state.imports.add(`import * as $ from 'ripple/internal/client'`);
      }
    }

    const metadata = { await: false };

    const body_statements = transform_body(node.body, {
      ...context,
      state: { ...context.state, component: node, metadata }
    });

    return b.function(
      node.id,
      [b.id('__anchor'), ...node.params.map((param) => context.visit(param, context.state))],
      b.block(
        metadata.await
          ? [b.stmt(b.call('$.async', b.thunk(b.block(body_statements), true)))]
          : body_statements
      )
    );
  },

  Component(node, context) {
    let prop_statements;

    if (!context.state.to_ts) {
      if (!context.state.imports.has(`import * as $ from 'ripple/internal/client'`)) {
        context.state.imports.add(`import * as $ from 'ripple/internal/client'`);
      }
    }

    const metadata = { await: false };

    if (context.state.to_ts) {
      const body_statements = [
        ...transform_body(node.body, {
          ...context,
          state: { ...context.state, component: node, metadata }
        })
      ];
```

```
      return b.function(node.id, node.params, b.block(body_statements));
    }

    if (node.params.length > 0) {
      let props = node.params[0];

      if (props.type === 'Identifier') {
        delete props.typeAnnotation;
      } else if (props.type === 'ObjectPattern') {
        const paths = extract_paths(props);

        for (const path of paths) {
          const name = path.node.name;
          const binding = context.state.scope.get(name);
          const key = b.key(name);

          if (binding !== null && !is_tracked_name(name)) {
            if (prop_statements === undefined) {
              prop_statements = [];
            }
            prop_statements.push(b.var(name, b.member(b.id('__props'), key)));
          }
        }
      }
    }

    const body_statements = [
      b.stmt(b.call('$.push_component')),
      ...transform_body(node.body, {
        ...context,
        state: { ...context.state, component: node, metadata }
      }),
      b.stmt(b.call('$.pop_component'))
    ];

    if (node.css !== null) {
      context.state.stylesheets.push(node.css);
    }

    return b.function(
      node.id,
      node.params.length > 0
        ? [
            b.id('__anchor'),
            node.params[0].type === 'Identifier' ? node.params[0] : b.id('__props'),
            b.id('__block')
          ]
        : [b.id('__anchor'), b.id('_'), b.id('__block')],
      b.block([
        ...(prop_statements ?? []),
        ...(metadata.await
          ? [b.stmt(b.call('$.async', b.thunk(b.block(body_statements), true)))]
          : body_statements)
      ])
    );
  },

  AssignmentExpression(node, context) {
    if (context.state.to_ts) {
```

```
      return context.next();
    }

    const left = node.left;

    if (
      left.type === 'MemberExpression' &&
      ((left.property.type === 'Identifier' && is_tracked_name(left.property.name)) ||
        left.computed)
    ) {
      return b.call(
        '$.set_property',
        context.visit(left.object),
        left.computed ? context.visit(left.property) : b.literal(left.property.name),
        visit_assignment_expression(node, context, build_assignment) ?? context.next(),
        b.id('__block')
      );
    }

    const visited = visit_assignment_expression(node, context, build_assignment) ?? context.next();

    if (
      left.type === 'MemberExpression' &&
      left.property.type === 'Identifier' &&
      left.property.name === '$length' &&
      !left.computed
    ) {
      return b.call('$.with_scope', b.id('__block'), b.thunk(visited));
    }

    return visited;
  },

  UpdateExpression(node, context) {
    if (context.state.to_ts) {
      context.next();
      return;
    }
    const argument = node.argument;

    if (
      argument.type === 'MemberExpression' &&
      ((argument.property.type === 'Identifier' && is_tracked_name(argument.property.name)) ||
        argument.computed)
    ) {
      return b.call(
        node.prefix ? '$.update_pre_property' : '$.update_property',
        context.visit(argument.object),
        argument.computed ? context.visit(argument.property) : b.literal(argument.property.name),
        b.id('__block'),
        node.operator === '--' ? b.literal(-1) : undefined
      );
    }

    const left = object(argument);
    const binding = context.state.scope.get(left.name);
    const transformers = left && binding?.transform;

    if (left === argument && transformers?.update) {
      // we don't need to worry about ownership_invalid_mutation here, because
```

```
      // we're not mutating but reassigning
      return transformers.update(node);
    }

    context.next();
  },

  ObjectExpression(node, context) {
    const properties = [];
    const tracked = [];

    for (const property of node.properties) {
      if (
        property.type === 'Property' &&
        !property.computed &&
        property.key.type === 'Identifier' &&
        property.kind === 'init' &&
        is_tracked_name(property.key.name)
      ) {
        tracked.push(b.literal(property.key.name));
        const metadata = { tracking: false, await: false };
        const tracked_property = context.visit(property, { ...context.state, metadata });

        if (metadata.tracking) {
          properties.push({
            ...tracked_property,
            value: b.call('$.computed_property', b.thunk(tracked_property.value), b.id('__block'))
          });
        } else {
          properties.push(tracked_property);
        }
      } else {
        properties.push(context.visit(property));
      }
    }

    if (tracked.length > 0) {
      return b.call('$.tracked_object', { ...node, properties }, b.array(tracked), b.id('__block'));
    }

    context.next();
  },

  ArrayExpression(node, context) {
    const elements = [];
    const tracked = [];
    let i = 0;

    for (const element of node.elements) {
      if (element === null) {
        elements.push(null);
      } else if (element.type === 'Element') {
        const metadata = { tracking: false, await: false };
        const tracked_element = context.visit(element, { ...context.state, metadata });

        if (metadata.tracking) {
          tracked.push(b.literal(i));
          elements.push(tracked_element);
        } else {
          elements.push(tracked_element);
```

```
        }
      } else if (element.type === 'SpreadElement') {
        const metadata = { tracking: false, await: false };
        const tracked_element = context.visit(element, { ...context.state, metadata });

        if (metadata.tracking) {
          tracked.push(b.spread(b.call('Object.keys', tracked_element.argument)));
          elements.push(tracked_element);
        } else {
          elements.push(tracked_element);
        }
      } else {
        const metadata = { tracking: false, await: false };
        elements.push(context.visit(element, { ...context.state, metadata }));
      }
      i++;
    }

    if (tracked.length > 0) {
      return b.call('$.tracked_object', { ...node, elements }, b.array(tracked), b.id('__block'));
    }

    context.next();
  },

  ForOfStatement(node, context) {
    if (!is_inside_component(context)) {
      context.next();
      return;
    }
    const is_controlled = node.is_controlled;

    // do only if not controller
    if (!is_controlled) {
      context.state.template.push('<!>');
    }

    const id = context.state.flush_node(is_controlled);
    const pattern = node.left.declarations[0].id;
    const body_scope = context.state.scopes.get(node.body);

    context.state.init.push(
      b.stmt(
        b.call(
          '$.for',
          id,
          b.thunk(context.visit(node.right)),
          b.arrow(
            [b.id('__anchor'), pattern],
            b.block(
              transform_body(node.body.body, {
                ...context,
                state: { ...context.state, scope: body_scope }
              })
            )
          ),
          b.literal(is_controlled ? IS_CONTROLLED : 0)
        )
      )
    );
```

```
  },

  IfStatement(node, context) {
    if (!is_inside_component(context)) {
      context.next();
      return;
    }
    context.state.template.push('<!>');

    const id = context.state.flush_node();
    const statements = [];

    const consequent_scope = context.state.scopes.get(node.consequent);
    const consequent = b.block(
      transform_body(node.consequent.body, {
        ...context,
        state: { ...context.state, scope: consequent_scope }
      })
    );
    const consequent_id = context.state.scope.generate('consequent');

    statements.push(b.var(b.id(consequent_id), b.arrow([b.id('__anchor')], consequent)));

    let alternate_id;

    if (node.alternate !== null) {
      const alternate_scope = context.state.scopes.get(node.alternate) || context.state.scope;
      let alternate_body = node.alternate.body;
      if (node.alternate.type === 'IfStatement') {
        alternate_body = [node.alternate];
      }
      const alternate = b.block(
        transform_body(alternate_body, {
          ...context,
          state: { ...context.state, scope: alternate_scope }
        })
      );
      alternate_id = context.state.scope.generate('alternate');
      statements.push(b.var(b.id(alternate_id), b.arrow([b.id('__anchor')], alternate)));
    }

    statements.push(
      b.stmt(
        b.call(
          '$.if',
          id,
          b.arrow(
            [b.id('__render')],
            b.block([
              b.if(
                context.visit(node.test),
                b.stmt(b.call(b.id('__render'), b.id(consequent_id))),
                alternate_id
                  ? b.stmt(
                      b.call(
                        b.id('__render'),
                        b.id(alternate_id),
                        node.alternate ? b.literal(false) : undefined
                      )
                    )
```

```
                : undefined
            )
          ])
        )
      )
    )
  );

  context.state.init.push(b.block(statements));
},

TryStatement(node, context) {
  if (!is_inside_component(context)) {
    context.next();
    return;
  }
  context.state.template.push('<!>');

  const id = context.state.flush_node();
  const metadata = { await: false };
  let body = transform_body(node.block.body, {
    ...context,
    state: { ...context.state, metadata }
  });

  if (metadata.await) {
    body = [b.stmt(b.call('$.async', b.thunk(b.block(body), true)))];
  }

  context.state.init.push(
    b.stmt(
      b.call(
        '$.try',
        id,
        b.arrow([b.id('__anchor')], b.block(body)),
        node.handler === null
          ? b.literal(null)
          : b.arrow(
              [b.id('__anchor'), ...(node.handler.param ? [node.handler.param] : [])],
              b.block(transform_body(node.handler.body.body, context))
            ),
        node.async === null
          ? undefined
          : b.arrow([b.id('__anchor')], b.block(transform_body(node.async.body, context)))
      )
    )
  );
},

AwaitExpression(node, context) {
  if (!is_inside_component(context)) {
    context.next();
  }

  if (context.state.metadata?.await === false) {
    context.state.metadata.await = true;
  }

  return b.call(b.await(b.call('$.resume_context', context.visit(node.argument))));
},
```

```
  JSXText(node) {
    const text = node.value;
    if (text.trim() === '') {
      return b.empty;
    }
    return b.literal(text);
  },

  JSXExpressionContainer(node, context) {
    const expression = context.visit(node.expression);
    if (expression.type === b.empty) {
      return b.empty;
    }
    return expression;
  },

  JSXIdentifier(node, context) {
    return context.visit(b.id(node.name));
  },

  JSXMemberExpression(node, context) {
    return b.member(context.visit(node.object), context.visit(node.property));
  },

  BinaryExpression(node, context) {
    return b.binary(node.operator, context.visit(node.left), context.visit(node.right));
  },

  JSXElement(node, context) {
    if (
      !context.state.imports.has(`import { jsx as _jsx, jsxs as _jsxs } from 'react/jsx-runtime'`)
    ) {
      context.state.imports.add(`import { jsx as _jsx, jsxs as _jsxs } from 'react/jsx-runtime'`);
    }

    const openingElement = node.openingElement;
    const name = openingElement.name;
    const props = b.object([]);
    const children = node.children
      .map((child) => context.visit(child, context.state))
      .filter((child) => child !== b.empty);

    if (children.length > 0) {
      props.properties.push(b.prop('init', b.id('children'), b.array(children)));
    }

    for (const attr of openingElement.attributes) {
      if (attr.type === 'JSXAttribute') {
        props.properties.push(
          b.prop('init', b.id(attr.name.name), context.visit(attr.value, context.state))
        );
      }
    }

    return b.call(
      children.length > 0 ? '_jsxs' : '_jsx',
      name.type === 'JSXIdentifier' && name.name[0] === name.name[0].toLowerCase()
        ? b.literal(name.name)
        : context.visit(name),
```

```
      props
    );
  },

  RenderFragment(node, context) {
    const identifer = node.expression.callee;

    context.state.template.push('<!>');

    const id = context.state.flush_node();

    context.state.init.push(
      b.stmt(
        b.call(
          context.visit(identifer),
          id,
          ...node.expression.arguments.map((arg) => context.visit(arg, context.state))
        )
      )
    );
  },

  BlockStatement(node, context) {
    const statements = [];

    for (const statement of node.body) {
      if (statement.type === 'Eval') {
        const finalizer = statement.finalizer;
        if (finalizer === null) {
          throw new Error('`eval` block should have been transformed');
        }

        for (const final_node of finalizer.body) {
          if (final_node.type === 'EmptyStatement') {
            continue;
          } else {
            statements.push(context.visit(final_node));
          }
        }
        continue;
      }
      statements.push(context.visit(statement));
    }

    return b.block(statements);
  },

  Program(node, context) {
    const statements = [];

    for (const statement of node.body) {
      if (statement.type === 'Eval') {
        const finalizer = statement.finalizer;
        if (finalizer === null) {
          throw new Error('`eval` block should have been transformed');
        }

        for (const final_node of finalizer.body) {
          if (final_node.type === 'EmptyStatement') {
            continue;
```

```
          } else {
            statements.push(context.visit(final_node));
          }
        }
        continue;
      }
      statements.push(context.visit(statement));
    }

    return { ...node, body: statements };
  }
};

/**
 * @param {Array<string | Expression>} items
 */
function join_template(items) {
  let quasi = b.quasi('');
  const template = b.template([quasi], []);

  /**
   * @param {Expression} expression
   */
  function push(expression) {
    if (expression.type === 'TemplateLiteral') {
      for (let i = 0; i < expression.expressions.length; i += 1) {
        const q = expression.quasis[i];
        const e = expression.expressions[i];

        quasi.value.cooked += /** @type {string} */ (q.value.cooked);
        push(e);
      }

      const last = /** @type {TemplateElement} */ (expression.quasis.at(-1));
      quasi.value.cooked += /** @type {string} */ (last.value.cooked);
    } else if (expression.type === 'Literal') {
      /** @type {string} */ (quasi.value.cooked) += expression.value;
    } else {
      template.expressions.push(expression);
      template.quasis.push((quasi = b.quasi('')));
    }
  }

  for (const item of items) {
    if (typeof item === 'string') {
      quasi.value.cooked += item;
    } else {
      push(item);
    }
  }

  for (const quasi of template.quasis) {
    quasi.value.raw = sanitize_template_string(/** @type {string} */ (quasi.value.cooked));
  }

  quasi.tail = true;

  return template;
}
```

```
function normalize_child(node, normalized) {
  if (node.type === 'EmptyStatement') {
    return;
  } else if (node.type === 'Eval') {
    const finalizer = node.finalizer;
    if (finalizer === null) {
      throw new Error('`eval` block should have been transformed');
    }

    for (const final_node of finalizer.body) {
      normalize_child(final_node, normalized);
    }
  } else {
    normalized.push(node);
  }
}

function transform_ts_child(node, context) {
  const { state, visit } = context;

  if (node.type === 'Text') {
    state.init.push(b.stmt(visit(node.expression, { ...state })));
  } else if (node.type === 'Element') {
    const type = node.id.name;
    const children = [];
    let has_children_props = false;

    const attributes = node.attributes.map((attr) => {
      if (attr.type === 'Attribute') {
        const metadata = { await: false };
        const name = visit(attr.name, { ...state, metadata });
        const value = visit(attr.value, { ...state, metadata });
        const jsx_name = b.jsx_id(name.name);
        if (name.name === '$children') {
          has_children_props = true;
        }
        jsx_name.loc = name.loc;

        return b.jsx_attribute(jsx_name, b.jsx_expression_container(value));
      } else {
        debugger;
      }
    });

    if (!node.selfClosing && !has_children_props && node.children.length > 0) {
      const is_dom_element = type[0].toLowerCase() === type[0];

      const component_scope = context.state.scopes.get(node);
      const thunk = b.thunk(
        b.block(
          transform_body(node.children, {
            ...context,
            state: { ...state, scope: component_scope }
          })
        )
      );

      if (is_dom_element) {
        children.push(b.jsx_expression_container(b.call(thunk)));
      } else {
```

```
        const children_name = context.state.scope.generate('fragment');
        const children_id = b.id(children_name);
        const jsx_id = b.jsx_id('$children');
        jsx_id.loc = node.id.loc;
        state.init.push(b.const(children_id, thunk));
        attributes.push(b.jsx_attribute(jsx_id, b.jsx_expression_container(children_id)));
      }
    }

    const opening_type = b.jsx_id(type);
    opening_type.loc = node.id.loc;

    let closing_type = undefined;

    if (!node.selfClosing) {
      closing_type = b.jsx_id(type);
      closing_type.loc = {
        start: {
          line: node.loc.end.line,
          column: node.loc.end.column - type.length - 1
        },
        end: {
          line: node.loc.end.line,
          column: node.loc.end.column - 1
        }
      };
    }

    state.init.push(
      b.stmt(b.jsx_element(opening_type, attributes, children, node.selfClosing, closing_type))
    );
  } else if (node.type === 'IfStatement') {
    const consequent_scope = context.state.scopes.get(node.consequent);
    const consequent = b.block(
      transform_body(node.consequent.body, {
        ...context,
        state: { ...context.state, scope: consequent_scope }
      })
    );

    let alternate;

    if (node.alternate !== null) {
      const alternate_scope = context.state.scopes.get(node.alternate) || context.state.scope;
      let alternate_body = node.alternate.body;
      if (node.alternate.type === 'IfStatement') {
        alternate_body = [node.alternate];
      }
      alternate = b.block(
        transform_body(alternate_body, {
          ...context,
          state: { ...context.state, scope: alternate_scope }
        })
      );
    }

    state.init.push(b.if(visit(node.test), consequent, alternate));
  } else if (node.type === 'ForOfStatement') {
    const body_scope = context.state.scopes.get(node.body);
    const body = b.block(
```

```
      transform_body(node.body.body, {
        ...context,
        state: { ...context.state, scope: body_scope }
      })
    );

    state.init.push(b.for_of(visit(node.left), visit(node.right), body, node.await));
  } else if (node.type === 'RenderFragment') {
    const identifer = node.expression.callee;

    state.init.push(
      b.stmt(
        b.call(
          context.visit(identifer),
          ...node.expression.arguments.map((arg) => context.visit(arg, context.state))
        )
      )
    );
  } else {
    throw new Error('TODO');
  }
}

function transform_children(children, { visit, state, root }) {
  const normalized = [];

  for (const node of children) {
    normalize_child(node, normalized);
  }

  const is_fragment =
    normalized.some(
      (node) =>
        node.type === 'IfStatement' ||
        node.type === 'TryStatement' ||
        node.type === 'ForOfStatement' ||
        node.type === 'RenderFragment' ||
        (node.type === 'Element' && node.id.name[0].toLowerCase() !== node.id.name[0])
    ) ||
    normalized.filter(
      (node) => node.type !== 'VariableDeclaration' && node.type !== 'EmptyStatement'
    ).length > 1;
  let initial = null;
  let prev = null;
  let template_id = null;

  const get_id = (node) => {
    return b.id(
      node.type == 'Element' && node.id.name[0].toLowerCase() === node.id.name[0]
        ? state.scope.generate(node.id.name)
        : node.type == 'Text'
        ? state.scope.generate('text')
        : state.scope.generate('node')
    );
  };

  const create_initial = (node) => {
    const id = is_fragment ? b.id(state.scope.generate('fragment')) : get_id(node);
    initial = id;
    template_id = state.scope.generate('root');
```

```
    state.setup.push(b.var(id, b.call(template_id)));
  };

  for (let i = normalized.length - 1; i >= 0; i--) {
    const child = normalized[i];
    const prev_child = normalized[i - 1];

    if (child.type === 'Text' && prev_child?.type === 'Text') {
      if (child.expression.type === 'Literal' && prev_child.expression.type === 'Literal') {
        prev_child.expression.value += child.expression.value;
      } else {
        prev_child.expression = b.binary('+', prev_child.expression, child.expression);
      }
      normalized.splice(i, 1);
    }
  }

  for (const node of normalized) {
    if (
      node.type === 'VariableDeclaration' ||
      node.type === 'ExpressionStatement' ||
      node.type === 'FunctionDeclaration' ||
      node.type === 'DebuggerStatement' ||
      node.type === 'ClassDeclaration'
    ) {
      const metadata = { await: false };
      state.init.push(visit(node, { ...state, metadata }));
      if (metadata.await) {
        state.init.push(b.if(b.call('$.aborted'), b.return(null)));
        if (state.metadata?.await === false) {
          state.metadata.await = true;
        }
      }
    } else if (state.to_ts) {
      transform_ts_child(node, { visit, state });
    } else {
      if (initial === null && root) {
        create_initial(node);
      }

      const current_prev = prev;
      let cached;
      const flush_node = (is_controlled) => {
        if (cached && !is_controlled) {
          return cached;
        } else if (current_prev !== null) {
          const id = get_id(node);
          state.setup.push(b.var(id, b.call('$.sibling', current_prev())));
          cached = id;
          return id;
        } else if (initial !== null) {
          if (is_fragment) {
            const id = get_id(node);
            state.setup.push(b.var(id, b.call('$.child_frag', initial)));
            cached = id;
            return id;
          }
          return initial;
        } else if (state.flush_node !== null) {
          if (is_controlled) {
```

```
            return state.flush_node();
          }

          const id = get_id(node);
          state.setup.push(b.var(id, b.call('$.child', state.flush_node())));
          cached = id;
          return id;
        } else {
          debugger;
        }
      };

      prev = flush_node;

      if (node.type === 'Element') {
        visit(node, { ...state, flush_node });
      } else if (node.type === 'Text') {
        const metadata = { tracking: false, await: false };
        const expression = visit(node.expression, { ...state, metadata });

        if (metadata.tracking) {
          state.template.push(' ');
          const id = flush_node();
          state.update.push(b.stmt(b.call('$.set_text', id, expression)));
        } else if (normalized.length === 1) {
          if (expression.type === 'Literal') {
            state.template.push(expression.value);
          } else {
            const id = state.flush_node();
            state.init.push(
              b.stmt(b.assignment('=', b.member(id, b.id('textContent')), expression))
            );
          }
        } else {
          // Handle Text nodes in fragments
          state.template.push(' ');
          const id = flush_node();
          state.update.push(b.stmt(b.call('$.set_text', id, expression)));
        }
      } else if (node.type === 'ForOfStatement') {
        const is_controlled = normalized.length === 1;
        node.is_controlled = is_controlled;
        visit(node, { ...state, flush_node });
      } else if (node.type === 'IfStatement') {
        const is_controlled = normalized.length === 1;
        node.is_controlled = is_controlled;
        visit(node, { ...state, flush_node });
      } else if (node.type === 'TryStatement') {
        const is_controlled = normalized.length === 1;
        node.is_controlled = is_controlled;
        visit(node, { ...state, flush_node });
      } else if (node.type === 'RenderFragment') {
        const is_controlled = normalized.length === 1;
        node.is_controlled = is_controlled;
        visit(node, { ...state, flush_node });
      } else {
        debugger;
      }
    }
  }
```

```
  if (root && initial !== null && template_id !== null) {
    const flags = is_fragment ? b.literal(TEMPLATE_FRAGMENT) : b.literal(0);
    state.final.push(b.stmt(b.call('$.append', b.id('__anchor'), initial)));
    state.hoisted.push(
      b.var(template_id, b.call('$.template', join_template(state.template), flags))
    );
  }
}

function transform_body(body, { visit, state }) {
  const body_state = {
    ...state,
    template: [],
    setup: [],
    init: [],
    update: [],
    final: [],
    metadata: state.metadata
  };

  transform_children(body, { visit, state: body_state, root: true });

  if (body_state.update.length > 0) {
    body_state.init.push(b.stmt(b.call('$.render', b.thunk(b.block(body_state.update)))));
  }

  return [...body_state.setup, ...body_state.init, ...body_state.final];
}

export function transform(filename, source, analysis, to_ts) {
  const state = {
    imports: new Set(),
    events: new Set(),
    template: null,
    hoisted: [],
    setup: null,
    init: null,
    update: null,
    final: null,
    flush_node: null,
    scope: analysis.scope,
    scopes: analysis.scopes,
    stylesheets: [],
    to_ts
  };

  const program = /** @type {ESTree.Program} */ (
    walk(/** @type {AST.SvelteNode} */ (analysis.ast), state, visitors)
  );

  for (const hoisted of state.hoisted) {
    program.body.unshift(hoisted);
  }

  for (const import_node of state.imports) {
    program.body.unshift(b.stmt(b.id(import_node)));
  }

  if (state.events.size > 0) {
```

```
    program.body.push(
      b.stmt(b.call('$.delegate', b.array(Array.from(state.events).map((name) => b.literal(name)))))
    );
  }

  const js = print(
    program,
    tsx({
      comments: analysis.ast.comments || []
    }),
    {
      sourceMapContent: source,
      sourceMapSource: path.basename(filename)
    }
  );

  const css = render_stylesheets(state.stylesheets);

  return {
    ast: program,
    js,
    css
  };
}
```

src/compiler/phases/3-transform/segments.js

```
import { decode } from '@jridgewell/sourcemap-codec';

export const defaultMappingData = {
 verification: true,
 completion: true,
 semantic: true,
 navigation: true
};

/**
 * Convert esrap SourceMap to Volar mappings using Svelte's approach
 * Based on: https://github.com/volarjs/svelte-language-tools/blob/master/packages/language-server/s
rc/language.ts#L45-L88
 * @param {object} source_map - esrap SourceMap object
 * @param {string} source - Original Ripple source
 * @param {string} generated_code - Generated TypeScript code
 * @returns {object} Object with code and mappings for Volar
 */
export function convert_source_map_to_mappings(source_map, source, generated_code) {
 const mappings = [];

 // Decode the VLQ mappings from esrap
 const decoded_mappings = decode(source_map.mappings);

 let generated_offset = 0;
 const generated_lines = generated_code.split('\n');

 // Process each line of generated code
 for (let generated_line = 0; generated_line < generated_lines.length; generated_line++) {
  const line = generated_lines[generated_line];
  const line_mappings = decoded_mappings[generated_line] || [];
```

```
  // Process mappings for this line
  for (const mapping of line_mappings) {
   const [generated_column, source_file_index, source_line, source_column] = mapping;

   // Skip mappings without source information
   if (source_file_index == null || source_line == null || source_column == null) {
    continue;
   }

   // Calculate source offset
   const source_lines = source.split('\n');
   let source_offset = 0;
   for (let i = 0; i < Math.min(source_line, source_lines.length - 1); i++) {
    source_offset += source_lines[i].length + 1; // +1 for newline
   }
   source_offset += source_column;

   // Calculate generated offset
   const current_generated_offset = generated_offset + generated_column;

   // Determine segment length (look ahead to next mapping or end of line)
   const next_mapping = line_mappings[line_mappings.indexOf(mapping) + 1];
   let segment_length = next_mapping ? next_mapping[0] - generated_column : Math.max(1, line.length
- generated_column);

   // Determine the actual segment content
   const generated_content = generated_code.substring(current_generated_offset, current_generated_of
fset + segment_length);
   const source_content = source.substring(source_offset, source_offset + segment_length);

   // Fix for $children mapping: when generated content is "$children",
   // it should only map to the component name in the source, not include attributes
   if (generated_content === '$children') {
    // Look for the component name in the source content
    const component_name_match = source_content.match(/^(\w+)/);
    if (component_name_match) {
     const component_name = component_name_match[1];
     segment_length = component_name.length;
    }
   }

   // Create Volar mapping with default mapping data
   mappings.push({
    sourceOffsets: [source_offset],
    generatedOffsets: [current_generated_offset],
    lengths: [segment_length],
    data: defaultMappingData
   });
  }

  // Add line length + 1 for newline (except for last line)
  generated_offset += line.length;
  if (generated_line < generated_lines.length - 1) {
   generated_offset += 1; // newline character
  }
 }

 return {
  code: generated_code,
```

```
  mappings
 };
}
```

src/compiler/phases/3-transform/stylesheet.js

```
import MagicString from 'magic-string';
import { walk } from 'zimmerframe';

const regex_css_browser_prefix = /^-((webkit)|(moz)|(o)|(ms))-/;

const is_keyframes_node = (node) => remove_css_prefix(node.name) === 'keyframes';

function remove_css_prefix(name) {
 return name.replace(regex_css_browser_prefix, '');
}

function remove_preceding_whitespace(end, state) {
 let start = end;
 while (/\s/.test(state.code.original[start - 1])) start--;
 if (start < end) state.code.remove(start, end);
}

function is_used(rule) {
 return rule.prelude.children.some((selector) => selector.metadata.used);
}

function is_in_global_block(path) {
 return path.some((node) => node.type === 'Rule' && node.metadata.is_global_block);
}

function remove_global_pseudo_class(selector, combinator, state) {
 if (selector.args === null) {
  let start = selector.start;
  if (combinator?.name === ' ') {
   // div :global.x becomes div.x
   while (/\s/.test(state.code.original[start - 1])) start--;
  }
  state.code.remove(start, selector.start + ':global'.length);
 } else {
  state.code
   .remove(selector.start, selector.start + ':global('.length)
   .remove(selector.end - 1, selector.end);
 }
}

function escape_comment_close(node, code) {
 let escaped = false;
 let in_comment = false;

 for (let i = node.start; i < node.end; i++) {
  if (escaped) {
   escaped = false;
  } else {
   const char = code.original[i];
   if (in_comment) {
    if (char === '*' && code.original[i + 1] === '/') {
     code.prependRight(++i, '\\');
```

```
      in_comment = false;
     }
    } else if (char === '\\') {
     escaped = true;
    } else if (char === '/' && code.original[++i] === '*') {
     in_comment = true;
    }
   }
  }
 }

 /**
  *  @param {AST.CSS.Rule} rule
  * @param {boolean} is_in_global_block
  */
 function is_empty(rule, is_in_global_block) {
  if (rule.metadata.is_global_block) {
   return rule.block.children.length === 0;
  }

  for (const child of rule.block.children) {
   if (child.type === 'Declaration') {
    return false;
   }

   if (child.type === 'Rule') {
    if ((is_used(child) || is_in_global_block) && !is_empty(child, is_in_global_block)) {
     return false;
    }
   }

   if (child.type === 'Atrule') {
    if (child.block === null || child.block.children.length > 0) return false;
   }
  }

  return true;
 }

 const visitors = {
  _: (node, context) => {
   context.state.code.addSourcemapLocation(node.start);
   context.state.code.addSourcemapLocation(node.end);
   context.next();
  },
  Atrule(node, { state, next, path }) {
   if (is_keyframes_node(node)) {
    let start = node.start + node.name.length + 1;
    while (state.code.original[start] === ' ') start += 1;
    let end = start;
    while (state.code.original[end] !== '{' && state.code.original[end] !== ' ') end += 1;

    if (node.prelude.startsWith('-global-')) {
     state.code.remove(start, start + 8);
    } else if (!is_in_global_block(path)) {
     state.code.prependRight(start, `${state.hash}-`);
    }

    return; // don't transform anything within
   }
```

```
 next();
},
Declaration(node, { state }) {
 const property = node.property && remove_css_prefix(node.property.toLowerCase());
 if (property === 'animation' || property === 'animation-name') {
  let index = node.start + node.property.length + 1;
  let name = '';

  while (index < state.code.original.length) {
   const character = state.code.original[index];

   if (regex_css_name_boundary.test(character)) {
    if (state.keyframes.includes(name)) {
     state.code.prependRight(index - name.length, `${state.hash}-`);
    }

    if (character === ';' || character === '}') {
     break;
    }

    name = '';
   } else {
    name += character;
   }

   index++;
  }
 }
},
Rule(node, { state, next, visit, path }) {
 if (is_empty(node, is_in_global_block(path))) {
  state.code.prependRight(node.start, '/* (empty) ');
  state.code.appendLeft(node.end, '*/');
  escape_comment_close(node, state.code);

  return;
 }

 if (!is_used(node) && !is_in_global_block(path)) {
  state.code.prependRight(node.start, '/* (unused) ');
  state.code.appendLeft(node.end, '*/');
  escape_comment_close(node, state.code);

  return;
 }

 if (node.metadata.is_global_block) {
  const selector = node.prelude.children[0];

  if (selector.children.length === 1 && selector.children[0].selectors.length === 1) {
   // `:global {...}`
   if (state.minify) {
    state.code.remove(node.start, node.block.start + 1);
    state.code.remove(node.block.end - 1, node.end);
   } else {
    state.code.prependRight(node.start, '/* ');
    state.code.appendLeft(node.block.start + 1, '*/');

    state.code.prependRight(node.block.end - 1, '/*');
```

```
      state.code.appendLeft(node.block.end, '*/');
     }

     // don't recurse into selectors but visit the body
     visit(node.block);
     return;
    }
   }

   next();
  },
  SelectorList(node, { state, next, path }) {
   // Only add comments if we're not inside a complex selector that itself is unused or a global bloc
k
   if (
    !is_in_global_block(path) &&
    !path.find((n) => n.type === 'ComplexSelector' && !n.metadata.used)
   ) {
    const children = node.children;
    let pruning = false;
    let prune_start = children[0].start;
    let last = prune_start;
    let has_previous_used = false;

    for (let i = 0; i < children.length; i += 1) {
     const selector = children[i];

     if (selector.metadata.used === pruning) {
      if (pruning) {
       let i = selector.start;
       while (state.code.original[i] !== ',') i--;

       if (state.minify) {
        state.code.remove(prune_start, has_previous_used ? i : i + 1);
       } else {
        state.code.appendRight(has_previous_used ? i : i + 1, '*/');
       }
      } else {
       if (i === 0) {
        if (state.minify) {
         prune_start = selector.start;
        } else {
         state.code.prependRight(selector.start, '/* (unused) ');
        }
       } else {
        if (state.minify) {
         prune_start = last;
        } else {
         state.code.overwrite(last, selector.start, ` /* (unused) `);
        }
       }
      }

      pruning = !pruning;
     }

     if (!pruning && selector.metadata.used) {
      has_previous_used = true;
     }
```

```
     last = selector.end;
    }

    if (pruning) {
     if (state.minify) {
      state.code.remove(prune_start, last);
     } else {
      state.code.appendLeft(last, '*/');
     }
    }
   }

   // if we're in a `:is(...)` or whatever, keep existing specificity bump state
   let specificity = state.specificity;

   // if this selector list belongs to a rule, require a specificity bump for the
   // first scoped selector but only if we're at the top level
   let parent = path.at(-1);
   if (parent?.type === 'Rule') {
    specificity = { bumped: false };

    /** @type {AST.CSS.Rule | null} */
    let rule = parent.metadata.parent_rule;

    while (rule) {
     if (rule.metadata.has_local_selectors) {
      specificity = { bumped: true };
      break;
     }
     rule = rule.metadata.parent_rule;
    }
   }

   next({ ...state, specificity });
  },
  ComplexSelector(node, context) {
   const before_bumped = context.state.specificity.bumped;

   for (const relative_selector of node.children) {
    if (relative_selector.metadata.is_global) {
     const global = /** @type {AST.CSS.PseudoClassSelector} */ (relative_selector.selectors[0]);
     remove_global_pseudo_class(global, relative_selector.combinator, context.state);

     if (
      node.metadata.rule?.metadata.parent_rule &&
      global.args === null &&
      relative_selector.combinator === null
     ) {
      // div { :global.x { ... } } becomes div { &.x { ... } }
      context.state.code.prependRight(global.start, '&');
     }
     continue;
    } else {
     // for any :global() or :global at the middle of compound selector
     for (const selector of relative_selector.selectors) {
      if (selector.type === 'PseudoClassSelector' && selector.name === 'global') {
       remove_global_pseudo_class(selector, null, context.state);
      }
     }
    }
```

```
   if (relative_selector.metadata.scoped) {
    if (relative_selector.selectors.length === 1) {
     // skip standalone :is/:where/& selectors
     const selector = relative_selector.selectors[0];
     if (
      selector.type === 'PseudoClassSelector' &&
      (selector.name === 'is' || selector.name === 'where')
     ) {
      continue;
     }
    }

    if (relative_selector.selectors.some((s) => s.type === 'NestingSelector')) {
     continue;
    }

    // for the first occurrence, we use a classname selector, so that every
    // encapsulated selector gets a +0-1-0 specificity bump. thereafter,
    // we use a `:where` selector, which does not affect specificity
    let modifier = context.state.selector;
    if (context.state.specificity.bumped) modifier = `:where(${modifier})`;

    context.state.specificity.bumped = true;

    let i = relative_selector.selectors.length;
    while (i--) {
     const selector = relative_selector.selectors[i];

     if (
      selector.type === 'PseudoElementSelector' ||
      selector.type === 'PseudoClassSelector'
     ) {
      if (selector.name !== 'root' && selector.name !== 'host') {
       if (i === 0) context.state.code.prependRight(selector.start, modifier);
      }
      continue;
     }

     if (selector.type === 'TypeSelector' && selector.name === '*') {
      context.state.code.update(selector.start, selector.end, modifier);
     } else {
      context.state.code.appendLeft(selector.end, modifier);
     }

     break;
    }
   }
  }

  context.next();

  context.state.specificity.bumped = before_bumped;
 },
 PseudoClassSelector(node, context) {
  if (node.name === 'is' || node.name === 'where' || node.name === 'has' || node.name === 'not') {
   context.next();
  }
 }
};
```

```
export function render_stylesheets(stylesheets) {
 let css = '';

 for (const stylesheet of stylesheets) {
  const code = new MagicString(stylesheet.source);
  const state = {
   code,
   hash: stylesheet.hash,
   selector: `.${stylesheet.hash}`,
   specificity: {
    bumped: false
   }
  };

  walk(stylesheet, state, visitors);
  css += code.toString();
 }

 return css;
}
```

src/compiler/errors.js

```
/**
 *
 * @param {string} message
 * @param {string} filename
 * @param {any} node
 */
export function error(message, filename, node) {
 let errorMessage = message;

 if (node && node.loc) {
  // Use GitHub-style range format: filename#L39C24-L39C32
  const startLine = node.loc.start.line;
  const startColumn = node.loc.start.column;
  const endLine = node.loc.end.line;
  const endColumn = node.loc.end.column;

  const rangeInfo = `${filename}#L${startLine}C${startColumn}-L${endLine}C${endColumn}`;
  errorMessage += ` (${rangeInfo})`;
 } else {
  errorMessage += ` (${filename})`;
 }

 throw new Error(errorMessage);
}
```

src/compiler/index.js

```
import { parse as parse_module } from './phases/1-parse/index.js';
import { analyze } from './phases/2-analyze/index.js';
```

```
import { transform } from './phases/3-transform/index.js';
import { convert_source_map_to_mappings } from './phases/3-transform/segments.js';

export function parse(source) {
 return parse_module(source);
}

export function compile(source, filename) {
 const ast = parse_module(source);
 const analysis = analyze(ast, filename);
 const result = transform(filename, source, analysis, false);

 return result;
}

export function compile_to_volar_mappings(source, filename) {
 // Parse and transform to get the esrap sourcemap
 const ast = parse_module(source);
 const analysis = analyze(ast, filename);
 const transformed = transform(filename, source, analysis, true);

 // Use Svelte's approach to convert esrap sourcemap to Volar mappings
 return convert_source_map_to_mappings(transformed.js.map, source, transformed.js.code);
}
```

src/compiler/scope.js

```
import is_reference from 'is-reference';
import { extract_identifiers, object, unwrap_pattern } from '../utils/ast.js';
import { walk } from 'zimmerframe';
import { is_reserved } from './utils.js';
import * as b from '../utils/builders.js';

export function create_scopes(ast, root, parent) {
 const scopes = new Map();
 const scope = new Scope(root, parent, false);
 scopes.set(ast, scope);

 const state = { scope };
 const references = [];
 const updates = [];

 function add_params(scope, params) {
  for (const param of params) {
   for (const node of extract_identifiers(param)) {
    scope.declare(node, 'normal', param.type === 'RestElement' ? 'rest_param' : 'param');
   }
  }
 }

 const create_block_scope = (node, { state, next }) => {
  const scope = state.scope.child(true);
  scopes.set(node, scope);

  next({ scope });
 };

 walk(ast, state, {
```

```
  // references
  Identifier(node, { path, state }) {
   const parent = path.at(-1);
   if (
    parent &&
    is_reference(node, /** @type {Node} */ (parent)) &&
    // TSTypeAnnotation, TSInterfaceDeclaration etc - these are normally already filtered out,
    // but for the migration they aren't, so we need to filter them out here
    // TODO -> once migration script is gone we can remove this check
    !parent.type.startsWith('TS')
   ) {
    references.push([state.scope, { node, path: path.slice() }]);
   }
  },

  AssignmentExpression(node, { state, next }) {
   updates.push([state.scope, node.left]);
   next();
  },

  UpdateExpression(node, { state, next }) {
   updates.push([state.scope, /** @type {Identifier | MemberExpression} */ (node.argument)]);
   next();
  },

  ImportDeclaration(node, { state }) {
   for (const specifier of node.specifiers) {
    state.scope.declare(specifier.local, 'normal', 'import', node);
   }
  },

  Component(node, { state, next }) {
   const scope = state.scope.child();
   scopes.set(node, scope);

   if (node.id) scope.declare(node.id, 'normal', 'component');

   add_params(scope, node.params);
   next({ scope });
  },

  Fragment(node, { state, next }) {
   const scope = state.scope.child();
   scopes.set(node, scope);

   if (node.id) scope.declare(node.id, 'normal', 'fragment');

   add_params(scope, node.params);
   next({ scope });
  },

  Element(node, { state, next }) {
   const scope = state.scope.child();
   scopes.set(node, scope);

   scope.declare(node, 'normal', 'element');

   next({ scope });
  },
```

```
  FunctionExpression(node, { state, next }) {
   const scope = state.scope.child();
   scopes.set(node, scope);

   if (node.id) scope.declare(node.id, 'normal', 'function');

   add_params(scope, node.params);
   next({ scope });
  },

  FunctionDeclaration(node, { state, next }) {
   if (node.id) state.scope.declare(node.id, 'normal', 'function', node);

   const scope = state.scope.child();
   scopes.set(node, scope);

   add_params(scope, node.params);
   next({ scope });
  },

  ArrowFunctionExpression(node, { state, next }) {
   const scope = state.scope.child();
   scopes.set(node, scope);

   add_params(scope, node.params);
   next({ scope });
  },

  ForStatement: create_block_scope,
  ForInStatement: create_block_scope,
  ForOfStatement: create_block_scope,
  SwitchStatement: create_block_scope,
  BlockStatement(node, context) {
   const parent = context.path.at(-1);
   if (
    parent?.type === 'FunctionDeclaration' ||
    parent?.type === 'FunctionExpression' ||
    parent?.type === 'ArrowFunctionExpression'
   ) {
    // We already created a new scope for the function
    context.next();
   } else {
    create_block_scope(node, context);
   }
  },

  Eval(node, context) {
   const finalizer = node.finalizer;

   if (finalizer !== null) {
    for (const node of finalizer.body) {
     context.visit(node);
    }
   }
  },

  ClassDeclaration(node, { state, next }) {
   if (node.id) state.scope.declare(node.id, 'normal', 'let', node);
   next();
  },
```

```
  VariableDeclaration(node, { state, path, next }) {
   for (const declarator of node.declarations) {
    /** @type {Binding[]} */
    const bindings = [];

    state.scope.declarators.set(declarator, bindings);

    for (const id of extract_identifiers(declarator.id)) {
     const binding = state.scope.declare(id, 'normal', node.kind, declarator.init);
     bindings.push(binding);
    }
   }

   next();
  },

  CatchClause(node, { state, next }) {
   if (node.param) {
    const scope = state.scope.child(true);
    scopes.set(node, scope);

    for (const id of extract_identifiers(node.param)) {
     scope.declare(id, 'normal', 'let');
    }

    next({ scope });
   } else {
    next();
   }
  }
 });

 for (const [scope, { node, path }] of references) {
  scope.reference(node, path);
 }

 for (const [scope, node] of updates) {
  for (const expression of unwrap_pattern(node)) {
   const left = object(expression);
   const binding = left && scope.get(left.name);

   if (binding !== null && left !== binding.node) {
    binding.updated = true;

    if (left === expression) {
     binding.reassigned = true;
    } else {
     binding.mutated = true;
    }
   }
  }
 }

 return {
  scope,
  scopes
 };
}
```

```
export class Scope {
 /** @type {ScopeRoot} */
 root;

 /**
     * The immediate parent scope
     * @type {Scope | null}
     */
 parent;

 /**
     * Whether or not `var` declarations are contained by this scope
     * @type {boolean}
     */
 #porous;

 /**
     * A map of every identifier declared by this scope, and all the
     * identifiers that reference it
     * @type {Map<string, Binding>}
     */
 declarations = new Map();

 /**
     * A map of declarators to the bindings they declare
     * @type {Map<VariableDeclarator | AST.LetDirective, Binding[]>}
     */
 declarators = new Map();

 /**
     * A set of all the names referenced with this scope
     * — useful for generating unique names
     * @type {Map<string, { node: Identifier; path: AST.SvelteNode[] }[]>}
     */
 references = new Map();

 /**
     * The scope depth allows us to determine if a state variable is referenced in its own scope,
     * which is usually an error. Block statements do not increase this value
     */
 function_depth = 0;

 /**
     * If tracing of reactive dependencies is enabled for this scope
     * @type {null | Expression}
     */
 tracing = null;

 /**
     *
     * @param {ScopeRoot} root
     * @param {Scope | null} parent
     * @param {boolean} porous
     */
 constructor(root, parent, porous) {
  this.root = root;
  this.parent = parent;
  this.#porous = porous;
  this.function_depth = parent ? parent.function_depth + (porous ? 0 : 1) : 0;
 }
```

```
 /**
     * @param {Identifier} node
     * @param {Binding['kind']} kind
     * @param {DeclarationKind} declaration_kind
     * @param {null | Expression | FunctionDeclaration | ClassDeclaration | ImportDeclaration | AST.
EachBlock | AST.SnippetBlock} initial
     * @returns {Binding}
     */
 declare(node, kind, declaration_kind, initial = null) {
  if (this.parent) {
   if (declaration_kind === 'var' && this.#porous) {
    return this.parent.declare(node, kind, declaration_kind);
   }

   if (declaration_kind === 'import') {
    return this.parent.declare(node, kind, declaration_kind, initial);
   }
  }

  if (this.declarations.has(node.name)) {
   // This also errors on var/function types, but that's arguably a good thing
   e.declaration_duplicate(node, node.name);
  }

  /** @type {Binding} */
  const binding = {
   node,
   references: [],
   initial,
   reassigned: false,
   mutated: false,
   updated: false,
   scope: this,
   kind,
   declaration_kind,
   is_called: false,
   prop_alias: null,
   metadata: null
  };

  this.declarations.set(node.name, binding);
  this.root.conflicts.add(node.name);
  return binding;
 }

 child(porous = false) {
  return new Scope(this.root, this, porous);
 }

 /**
     * @param {string} preferred_name
     * @returns {string}
     */
 generate(preferred_name) {
  if (this.#porous) {
   return /** @type {Scope} */ (this.parent).generate(preferred_name);
  }

  preferred_name = preferred_name.replace(/[^a-zA-Z0-9_$]/g, '_').replace(/^[0-9]/, '_');
```

```
  let name = preferred_name;
  let n = 1;

  while (
   this.references.has(name) ||
   this.declarations.has(name) ||
   this.root.conflicts.has(name) ||
   is_reserved(name)
  ) {
   name = `${preferred_name}_${n++}`;
  }

  this.references.set(name, []);
  this.root.conflicts.add(name);
  return name;
 }

 /**
     * @param {string} name
     * @returns {Binding | null}
     */
 get(name) {
  return this.declarations.get(name) ?? this.parent?.get(name) ?? null;
 }

 /**
     * @param {VariableDeclarator | AST.LetDirective} node
     * @returns {Binding[]}
     */
 get_bindings(node) {
  const bindings = this.declarators.get(node);
  if (!bindings) {
   throw new Error('No binding found for declarator');
  }
  return bindings;
 }

 /**
     * @param {string} name
     * @returns {Scope | null}
     */
 owner(name) {
  return this.declarations.has(name) ? this : this.parent && this.parent.owner(name);
 }

 /**
     * @param {Identifier} node
     * @param {AST.SvelteNode[]} path
     */
 reference(node, path) {
  path = [...path]; // ensure that mutations to path afterwards don't affect this reference
  let references = this.references.get(node.name);

  if (!references) this.references.set(node.name, (references = []));

  references.push({ node, path });

  const binding = this.declarations.get(node.name);
  if (binding) {
   binding.references.push({ node, path });
```

```
   } else if (this.parent) {
    this.parent.reference(node, path);
   } else {
    // no binding was found, and this is the top level scope,
    // which means this is a global
    this.root.conflicts.add(node.name);
   }
  }
}

export class ScopeRoot {
 /** @type {Set<string>} */
 conflicts = new Set();

 /**
     * @param {string} preferred_name
     */
 unique(preferred_name) {
  preferred_name = preferred_name.replace(/[^a-zA-Z0-9_$]/g, '_');
  let final_name = preferred_name;
  let n = 1;

  while (this.conflicts.has(final_name)) {
   final_name = `${preferred_name}_${n++}`;
  }

  this.conflicts.add(final_name);
  const id = b.id(final_name);
  return id;
 }
}
```

src/compiler/utils.js

```
import { build_assignment_value } from '../utils/ast.js';
import * as b from '../utils/builders.js';

const regex_return_characters = /\r/g;

const RESERVED_WORDS = [
 'arguments',
 'await',
 'break',
 'case',
 'catch',
 'class',
 'const',
 'continue',
 'debugger',
 'default',
 'delete',
 'do',
 'else',
 'enum',
 'eval',
 'export',
 'extends',
 'false',
```

```
  'finally',
  'for',
  'function',
  'if',
  'implements',
  'import',
  'in',
  'instanceof',
  'interface',
  'let',
  'new',
  'null',
  'package',
  'private',
  'protected',
  'public',
  'return',
  'static',
  'super',
  'switch',
  'this',
  'throw',
  'true',
  'try',
  'typeof',
  'var',
  'void',
  'while',
  'with',
  'yield'
];

export function is_reserved(word) {
  return RESERVED_WORDS.includes(word);
}

/**
 * Attributes that are boolean, i.e. they are present or not present.
 */
const DOM_BOOLEAN_ATTRIBUTES = [
  'allowfullscreen',
  'async',
  'autofocus',
  'autoplay',
  'checked',
  'controls',
  'default',
  'disabled',
  'formnovalidate',
  'hidden',
  'indeterminate',
  'inert',
  'ismap',
  'loop',
  'multiple',
  'muted',
  'nomodule',
  'novalidate',
  'open',
  'playsinline',
```

```
 'readonly',
 'required',
 'reversed',
 'seamless',
 'selected',
 'webkitdirectory',
 'defer',
 'disablepictureinpicture',
 'disableremoteplayback'
];

export function is_boolean_attribute(name) {
 return DOM_BOOLEAN_ATTRIBUTES.includes(name);
}

const DOM_PROPERTIES = [
 ...DOM_BOOLEAN_ATTRIBUTES,
 'formNoValidate',
 'isMap',
 'noModule',
 'playsInline',
 'readOnly',
 'value',
 'volume',
 'defaultValue',
 'defaultChecked',
 'srcObject',
 'noValidate',
 'allowFullscreen',
 'disablePictureInPicture',
 'disableRemotePlayback'
];

export function is_dom_property(name) {
 return DOM_PROPERTIES.includes(name);
}

/** List of Element events that will be delegated */
const DELEGATED_EVENTS = [
 'beforeinput',
 'click',
 'change',
 'dblclick',
 'contextmenu',
 'focusin',
 'focusout',
 'input',
 'keydown',
 'keyup',
 'mousedown',
 'mousemove',
 'mouseout',
 'mouseover',
 'mouseup',
 'pointerdown',
 'pointermove',
 'pointerout',
 'pointerover',
 'pointerup',
 'touchend',
```

```
 'touchmove',
 'touchstart'
];

export function is_delegated(event_name) {
 return DELEGATED_EVENTS.includes(event_name);
}

const PASSIVE_EVENTS = ['touchstart', 'touchmove'];

export function is_passive_event(name) {
 return PASSIVE_EVENTS.includes(name);
}

export function is_event_attribute(attr) {
 return attr.startsWith('on') && attr.length > 2 && attr[2] === attr[2].toUpperCase();
}

const unhoisted = { hoisted: false };

export function get_delegated_event(event_name, handler, state) {
 // Handle delegated event handlers. Bail out if not a delegated event.
 if (!handler || !is_delegated(event_name)) {
  return null;
 }

 /** @type {FunctionExpression | FunctionDeclaration | ArrowFunctionExpression | null} */
 let target_function = null;
 let binding = null;

 if (handler.type === 'ArrowFunctionExpression' || handler.type === 'FunctionExpression') {
  target_function = handler;
 } else if (handler.type === 'Identifier') {
  binding = state.scope.get(handler.name);

  if (state.analysis.module.scope.references.has(handler.name)) {
   // If a binding with the same name is referenced in the module scope (even if not declared ther
e), bail out
   return unhoisted;
  }

  if (binding != null) {
   for (const { path } of binding.references) {
    const parent = path.at(-1);
    if (parent === undefined) return unhoisted;

    const grandparent = path.at(-2);

    /** @type {AST.RegularElement | null} */
    let element = null;
    /** @type {string | null} */
    let event_name = null;
    if (
     parent.type === 'ExpressionTag' &&
     grandparent?.type === 'Attribute' &&
     is_event_attribute(grandparent)
    ) {
     element = /** @type {AST.RegularElement} */ (path.at(-3));
     const attribute = /** @type {AST.Attribute} */ (grandparent);
     event_name = get_attribute_event_name(attribute.name);
```

```
    }

    if (element && event_name) {
     if (
      element.type !== 'Element' ||
      element.metadata.has_spread ||
      !is_delegated(event_name)
     ) {
      return unhoisted;
     }
    } else if (parent.type !== 'FunctionDeclaration' && parent.type !== 'VariableDeclarator') {
     return unhoisted;
    }
   }
  }

  // If the binding is exported, bail out
  if (state.analysis.exports.find((node) => node.name === handler.name)) {
   return unhoisted;
  }

  if (binding !== null && binding.initial !== null && !binding.updated && !binding.is_called) {
   const binding_type = binding.initial.type;

   if (
    binding_type === 'ArrowFunctionExpression' ||
    binding_type === 'FunctionDeclaration' ||
    binding_type === 'FunctionExpression'
   ) {
    target_function = binding.initial;
   }
  }
 }

 // If we can't find a function, or the function has multiple parameters, bail out
 if (target_function == null || target_function.params.length > 1) {
  return unhoisted;
 }

 const visited_references = new Set();
 const scope = target_function.metadata.scope;
 for (const [reference] of scope.references) {
  // Bail out if the arguments keyword is used or $host is referenced
  if (reference === 'arguments') return unhoisted;

  const binding = scope.get(reference);
  const local_binding = state.scope.get(reference);

  // If we are referencing a binding that is shadowed in another scope then bail out.
  if (local_binding !== null && binding !== null && local_binding.node !== binding.node) {
   return unhoisted;
  }

  if (
   binding !== null &&
   // Bail out if the the binding is a rest param
   (binding.declaration_kind === 'rest_param' || // or any normal not reactive bindings that are mut
ated.
    // Bail out if we reference anything from the EachBlock (for now) that mutates in non-runes mod
e,
```

```
    (binding.kind === 'normal' && binding.updated))
  ) {
   return unhoisted;
  }
  visited_references.add(reference);
 }

 return { hoisted: true, function: target_function };
}

function get_hoisted_params(node, context) {
 const scope = context.state.scope;

 /** @type {Identifier[]} */
 const params = [];

 /**
    * We only want to push if it's not already present to avoid name clashing
    * @param {Identifier} id
    */
 function push_unique(id) {
  if (!params.find((param) => param.name === id.name)) {
   params.push(id);
  }
 }

 for (const [reference] of scope.references) {
  let binding = scope.get(reference);

  if (binding !== null && !scope.declarations.has(reference) && binding.initial !== node) {
   if (binding.kind === 'prop') {
    debugger;
   } else if (
    // imports don't need to be hoisted
    binding.declaration_kind !== 'import'
   ) {
    // create a copy to remove start/end tags which would mess up source maps
    push_unique(b.id(binding.node.name));
   }
  }
 }
 return params;
}

export function build_hoisted_params(node, context) {
 const hoisted_params = get_hoisted_params(node, context);
 node.metadata.hoisted_params = hoisted_params;

 /** @type {Pattern[]} */
 const params = [];

 if (node.params.length === 0) {
  if (hoisted_params.length > 0) {
   // For the event object
   params.push(b.id(context.state.scope.generate('_')));
  }
 } else {
  for (const param of node.params) {
   params.push(/** @type {Pattern} */ (context.visit(param)));
  }
```

```
  }

  params.push(...hoisted_params, b.id('__block'));
  return params;
}

export function is_inside_component(context) {
  for (let i = context.path.length - 1; i >= 0; i -= 1) {
    const context_node = context.path[i];
    const type = context_node.type;

    if (
      type === 'FunctionExpression' ||
      type === 'ArrowFunctionExpression' ||
      type === 'FunctionDeclaration'
    ) {
      return false;
    }
    if (type === 'Component') {
      return true;
    }
  }
  return false;
}

export function is_inside_call_expression(context) {
  for (let i = context.path.length - 1; i >= 0; i -= 1) {
    const context_node = context.path[i];
    const type = context_node.type;

    if (
      type === 'FunctionExpression' ||
      type === 'ArrowFunctionExpression' ||
      type === 'FunctionDeclaration'
    ) {
      return false;
    }
    if (type === 'CallExpression') {
      return true;
    }
  }
  return false;
}

export function is_tracked_name(name) {
  return (
    typeof name === 'string' &&
    name.startsWith('$') &&
    name.length > 1 &&
    name[1] !== '$' &&
    name !== '$length'
  );
}

export function is_svelte_import(callee, context) {
  if (callee.type === 'Identifier') {
    const binding = context.state.scope.get(callee.name);

    return (
      binding?.declaration_kind === 'import' &&
```

```
    binding.initial.source.type === 'Literal' &&
    binding.initial.source.value === 'ripple'
   );
  }

  return false;
}

export function is_declared_within_component(node, context) {
  const component = context.path.find((n) => n.type === 'Component');

  if (node.type === 'Identifier' && component) {
   const binding = context.state.scope.get(node.name);
   const component_scope = context.state.scopes.get(component);

   if (binding !== null && component_scope !== null) {
    let scope = binding.scope;

    while (scope !== null) {
     if (scope === component_scope) {
      return true;
     }
     scope = scope.parent;
    }
   }
  }

  return false;
}

function is_non_coercive_operator(operator) {
  return ['=', '||=', '&&=', '??='].includes(operator);
}

export function visit_assignment_expression(node, context, build_assignment) {
  if (
   node.left.type === 'ArrayPattern' ||
   node.left.type === 'ObjectPattern' ||
   node.left.type === 'RestElement'
  ) {
   const value = /** @type {Expression} */ (context.visit(node.right));
   const should_cache = value.type !== 'Identifier';
   const rhs = should_cache ? b.id('$$value') : value;

   let changed = false;

   const assignments = extract_paths(node.left).map((path) => {
    const value = path.expression?.(rhs);

    let assignment = build_assignment('=', path.node, value, context);
    if (assignment !== null) changed = true;

    return (
     assignment ??
     b.assignment(
      '=',
      /** @type {Pattern} */ (context.visit(path.node)),
      /** @type {Expression} */ (context.visit(value))
     )
    );
```

```
  });

  if (!changed) {
   // No change to output -> nothing to transform -> we can keep the original assignment
   return null;
  }

  const is_standalone = /** @type {Node} */ (context.path.at(-1)).type.endsWith('Statement');
  const sequence = b.sequence(assignments);

  if (!is_standalone) {
   // this is part of an expression, we need the sequence to end with the value
   sequence.expressions.push(rhs);
  }

  if (should_cache) {
   // the right hand side is a complex expression, wrap in an IIFE to cache it
   const iife = b.arrow([rhs], sequence);

   const iife_is_async =
    is_expression_async(value) ||
    assignments.some((assignment) => is_expression_async(assignment));

   return iife_is_async ? b.await(b.call(b.async(iife), value)) : b.call(iife, value);
  }

  return sequence;
 }

 if (node.left.type !== 'Identifier' && node.left.type !== 'MemberExpression') {
  throw new Error(`Unexpected assignment type ${node.left.type}`);
 }

 return build_assignment(node.operator, node.left, node.right, context);
}

export function build_assignment(operator, left, right, context) {
 let object = left;

 while (object.type === 'MemberExpression') {
  // @ts-expect-error
  object = object.object;
 }

 if (object.type !== 'Identifier') {
  return null;
 }

 const binding = context.state.scope.get(object.name);
 if (!binding) return null;

 const transform = binding.transform;

 const path = context.path.map((node) => node.type);

 // reassignment
 if (object === left && transform?.assign) {
  let value = /** @type {Expression} */ (
   context.visit(build_assignment_value(operator, left, right))
  );
```

```
  return transform.assign(object, value);
 }

 // mutation
 if (transform?.mutate) {
  return transform.mutate(
   object,
   b.assignment(
    operator,
    /** @type {Pattern} */ (context.visit(left)),
    /** @type {Expression} */ (context.visit(right))
   )
  );
 }

 return null;
}

const ATTR_REGEX = /[&"<]/g;
const CONTENT_REGEX = /[&<]/g;

export function escape_html(value, is_attr) {
 const str = String(value ?? '');

 const pattern = is_attr ? ATTR_REGEX : CONTENT_REGEX;
 pattern.lastIndex = 0;

 let escaped = '';
 let last = 0;

 while (pattern.test(str)) {
  const i = pattern.lastIndex - 1;
  const ch = str[i];
  escaped += str.substring(last, i) + (ch === '&' ? '&amp;' : ch === '"' ? '&quot;' : '&lt;');
  last = i + 1;
 }

 return escaped + str.substring(last);
}

export function hash(str) {
 str = str.replace(regex_return_characters, '');
 let hash = 5381;
 let i = str.length;

 while (i--) hash = ((hash << 5) - hash) ^ str.charCodeAt(i);
 return (hash >>> 0).toString(36);
}
```

src/runtime/internal/client/blocks.js

```
import {
 BLOCK_HAS_RUN,
 BRANCH_BLOCK,
 COMPUTED,
 CONTAINS_TEARDOWN,
```

```
 DESTROYED,
 EFFECT_BLOCK,
 PAUSED,
 RENDER_BLOCK,
 ROOT_BLOCK,
 TRY_BLOCK
} from './constants';
import { next_sibling } from './operations';
import {
 active_block,
 active_component,
 active_reaction,
 run_block,
 run_teardown,
 schedule_update
} from './runtime';
import { suspend } from './try';

export function user_effect(fn) {
 if (active_block === null) {
  throw new Error('effect() must be called within an active context, such as a component or effec
t');
 }

 var component = active_component;
 if (component !== null && !component.m) {
  var e = (component.e ??= []);
  e.push({
   b: active_block,
   fn,
   r: active_reaction
  });

  return;
 }

 return block(EFFECT_BLOCK, fn);
}

export function effect(fn) {
 return block(EFFECT_BLOCK, fn);
}

export function render(fn, flags = 0) {
 return block(RENDER_BLOCK | flags, fn);
}

export function branch(fn, flags = 0) {
 return block(BRANCH_BLOCK | flags, fn);
}

export function async(fn) {
 return block(BRANCH_BLOCK, async () => {
  const unsuspend = suspend();
  await fn();
  unsuspend();
 });
}

export function root(fn) {
```

```
 return block(ROOT_BLOCK, fn);
}

export function create_try_block(fn, state) {
 return block(TRY_BLOCK, fn, state);
}

function push_block(block, parent_block) {
 var parent_last = parent_block.last;
 if (parent_last === null) {
  parent_block.last = parent_block.first = block;
 } else {
  parent_last.next = block;
  block.prev = parent_last;
  parent_block.last = block;
 }
}

export function block(flags, fn, state = null) {
 var block = {
  c: active_component,
  d: null,
  first: null,
  f: flags,
  fn,
  last: null,
  next: null,
  p: active_block,
  prev: null,
  s: state,
  t: null
 };

 if (active_reaction !== null && (active_reaction.f & COMPUTED) !== 0) {
  (active_reaction.blocks ??= []).push(block);
 }

 if (active_block !== null) {
  push_block(block, active_block);
 }

 if ((flags & EFFECT_BLOCK) !== 0) {
  schedule_update(block);
 } else {
  run_block(block);
  block.f ^= BLOCK_HAS_RUN;
 }

 return block;
}

export function destroy_block_children(parent, remove_dom = false) {
 var block = parent.first;
 parent.first = parent.last = null;

 if ((parent.f & CONTAINS_TEARDOWN) !== 0) {
  while (block !== null) {
   var next = block.next;
   destroy_block(block, remove_dom);
   block = next;
```

```
  }
 }
}

export function destroy_non_branch_children(parent, remove_dom = false) {
 var block = parent.first;

 if (
  (parent.f & CONTAINS_TEARDOWN) === 0 &&
  parent.first !== null &&
  (parent.first.f & BRANCH_BLOCK) === 0
 ) {
  parent.first = parent.last = null;
 } else {
  while (block !== null) {
   var next = block.next;
   if ((block.f & BRANCH_BLOCK) === 0) {
    destroy_block(block, remove_dom);
   }
   block = next;
  }
 }
}

export function unlink_block(block) {
 var parent = block.p;
 var prev = block.prev;
 var next = block.next;

 if (prev !== null) prev.next = next;
 if (next !== null) next.prev = prev;

 if (parent !== null) {
  if (parent.first === block) parent.first = next;
  if (parent.last === block) parent.last = prev;
 }
}

export function pause_block(block) {
 if ((block.f & PAUSED) !== 0) {
  return;
 }
 block.f ^= PAUSED;

 var child = block.first;

 while (child !== null) {
  var next = child.next;
  pause_block(child);
  child = next;
 }

 run_teardown(block);
}

export function resume_block(block) {
 if ((block.f & PAUSED) === 0) {
  return;
 }
 block.f ^= PAUSED;
```

```
 if (is_block_dirty(block)) {
  schedule_update(block);
 }

 var child = block.first;

 while (child !== null) {
  var next = child.next;
  resume_block(child);
  child = next;
 }
}

export function is_destroyed(target_block) {
 var block = target_block;

 while (block !== null) {
  var flags = block.f;

  if ((flags & DESTROYED) !== 0) {
   return true;
  }
  if ((flags & ROOT_BLOCK) !== 0) {
   return false;
  }
  block = block.p;
 }
 return true;
}

export function destroy_block(block, remove_dom = true) {
 block.f ^= DESTROYED;

 var removed = false;

 if (remove_dom && (block.f & (BRANCH_BLOCK | ROOT_BLOCK)) !== 0) {
  var node = block.s.start;
  var end = block.s.end;

  while (node !== null) {
   var next = node === end ? null : next_sibling(node);

   node.remove();
   node = next;
  }

  removed = true;
 }

 destroy_block_children(block, remove_dom && !removed);

 run_teardown(block);

 var parent = block.p;

 // If the parent doesn't have any children, then skip this work altogether
 if (parent !== null && parent.first !== null) {
  unlink_block(block);
 }
```

```
 block.fn = block.s = block.d = block.p = null;
}
```

src/runtime/internal/client/constants.js

```
export var ROOT_BLOCK = 1 << 1;
export var RENDER_BLOCK = 1 << 2;
export var EFFECT_BLOCK = 1 << 3;
export var BRANCH_BLOCK = 1 << 4;
export var FOR_BLOCK = 1 << 5;
export var TRY_BLOCK = 1 << 6;
export var IF_BLOCK = 1 << 7;
export var ASYNC_BLOCK = 1 << 8;
export var COMPAT_BLOCK = 1 << 9;
export var CONTAINS_UPDATE = 1 << 10;
export var CONTAINS_TEARDOWN = 1 << 11;
export var BLOCK_HAS_RUN = 1 << 12;
export var TRACKED = 1 << 13;
export var COMPUTED = 1 << 14;
export var DEFERRED = 1 << 15;
export var PAUSED = 1 << 16;
export var DESTROYED = 1 << 17;

export var LOGIC_BLOCK = FOR_BLOCK | IF_BLOCK | TRY_BLOCK;

export var UNINITIALIZED = Symbol();
export var TRACKED_OBJECT = Symbol();
export var COMPUTED_PROPERTY = Symbol();
```

src/runtime/internal/client/events.js

```
import { is_passive_event } from '../../../compiler/utils';
import {
 active_block,
 active_reaction,
 set_active_block,
 set_active_reaction,
 set_tracking,
 tracking
} from './runtime';
import { array_from, define_property, is_array } from './utils';

/** @type {Set<string>} */
var all_registered_events = new Set();

/** @type {Set<(events: Array<string>) => void>} */
var root_event_handles = new Set();

export function handle_event_propagation(event) {
 var handler_element = this;
 var owner_document = /** @type {Node} */ (handler_element).ownerDocument;
 var event_name = event.type;
 var path = event.composedPath?.() || [];
 var current_target = /** @type {null | Element} */ (path[0] || event.target);
```

```
 // composedPath contains list of nodes the event has propagated through.
 // We check __root to skip all nodes below it in case this is a
 // parent of the __root node, which indicates that there's nested
 // mounted apps. In this case we don't want to trigger events multiple times.
 var path_idx = 0;

 // @ts-expect-error is added below
 var handled_at = event.__root;

 if (handled_at) {
  var at_idx = path.indexOf(handled_at);
  if (
   at_idx !== -1 &&
   (handler_element === document || handler_element === /** @type {any} */ (window))
  ) {
   // This is the fallback document listener or a window listener, but the event was already handled
   // -> ignore, but set handle_at to document/window so that we're resetting the event
   // chain in case someone manually dispatches the same event object again.
   // @ts-expect-error
   event.__root = handler_element;
   return;
  }

  // We're deliberately not skipping if the index is higher, because
  // someone could create an event programmatically and emit it multiple times,
  // in which case we want to handle the whole propagation chain properly each time.
  // (this will only be a false negative if the event is dispatched multiple times and
  // the fallback document listener isn't reached in between, but that's super rare)
  var handler_idx = path.indexOf(handler_element);
  if (handler_idx === -1) {
   // handle_idx can theoretically be -1 (happened in some JSDOM testing scenarios with an event lis
tener on the window object)
   // so guard against that, too, and assume that everything was handled at this point.
   return;
  }

  if (at_idx <= handler_idx) {
   path_idx = at_idx;
  }
 }

 current_target = /** @type {Element} */ (path[path_idx] || event.target);
 // there can only be one delegated event per element, and we either already handled the current tar
get,
 // or this is the very first target in the chain which has a non-delegated listener, in which case
it's safe
 // to handle a possible delegated event on it later (through the root delegation listener for examp
le).
 if (current_target === handler_element) return;

 // Proxy currentTarget to correct target
 define_property(event, 'currentTarget', {
  configurable: true,
  get() {
   return current_target || owner_document;
  }
 });

 var previous_block = active_block;
```

```
 var previous_reaction = active_reaction;
 var previous_tracking = tracking;

 set_active_block(null);
 set_active_reaction(null);
 set_tracking(false);

 try {
  /**
         * @type {unknown}
         */
  var throw_error;
  /**
         * @type {unknown[]}
         */
  var other_errors = [];

  while (current_target !== null) {
   /** @type {null | Element} */
   var parent_element =
    current_target.assignedSlot ||
    current_target.parentNode ||
    /** @type {any} */ (current_target).host ||
    null;

   try {
    // @ts-expect-error
    var delegated = current_target['__' + event_name];

    if (delegated !== undefined && !(/** @type {any} */ (current_target).disabled)) {
     if (is_array(delegated)) {
      var [fn, block, ...data] = delegated;
      fn.apply(current_target, [event, ...data, block]);
     } else {
      delegated.call(current_target, event);
     }
    }
   } catch (error) {
    if (throw_error) {
     other_errors.push(error);
    } else {
     throw_error = error;
    }
   }
   if (event.cancelBubble || parent_element === handler_element || parent_element === null) {
    break;
   }
   current_target = parent_element;
  }

  if (throw_error) {
   for (let error of other_errors) {
    // Throw the rest of the errors, one-by-one on a microtask
    queueMicrotask(() => {
     throw error;
    });
   }
   throw throw_error;
  }
 } finally {
```

```
  set_active_block(previous_block);
  // @ts-expect-error is used above
  event.__root = handler_element;
  // @ts-ignore remove proxy on currentTarget
  delete event.currentTarget;
  set_active_block(previous_block);
  set_active_reaction(previous_reaction);
  set_tracking(previous_tracking);
 }
}

function create_event(event_name, dom, handler, options = {}) {
 var block = active_block;

 function target_handler(/** @type {Event} */ event) {
  var previous_block = active_block;
  var previous_reaction = active_reaction;
  var previous_tracking = tracking;

  try {
   set_active_block(null);
   set_active_reaction(null);
   set_tracking(false);

   if (!options.capture) {
    // Only call in the bubble phase, else delegated events would be called before the capturing eve
nts
    handle_event_propagation.call(dom, event);
   }
   if (!event.cancelBubble) {
    return handler?.call(this, event);
   }
  } finally {
   set_active_block(previous_block);
   set_active_reaction(previous_reaction);
   set_tracking(previous_tracking);
  }
 }

 dom.addEventListener(event_name, target_handler, options);

 return target_handler;
}

export function event(event_name, dom, handler, capture, passive) {
 var options = { capture, passive };
 create_event(event_name, dom, handler, options);
}

export function delegate(events) {
 for (var i = 0; i < events.length; i++) {
  all_registered_events.add(events[i]);
 }

 for (var fn of root_event_handles) {
  fn(events);
 }
}

export function handle_root_events(target) {
```

```
 var registered_events = new Set();

 var event_handle = (events) => {
  for (var i = 0; i < events.length; i++) {
   var event_name = events[i];

   if (registered_events.has(event_name)) continue;
   registered_events.add(event_name);

   var passive = is_passive_event(event_name);

   target.addEventListener(event_name, handle_event_propagation, { passive });
  }
 };

 event_handle(array_from(all_registered_events));
 root_event_handles.add(event_handle);

 return () => {
  for (var event_name of registered_events) {
   target.removeEventListener(event_name, handle_event_propagation);
  }
  root_event_handles.delete(event_handle);
 };
}
```

src/runtime/internal/client/for.js

```
import { IS_CONTROLLED } from '../../../constants';
import { get_all_elements } from '../../array';
import { branch, destroy_block, destroy_block_children, render } from './blocks';
import { FOR_BLOCK, TRACKED_OBJECT } from './constants';
import { create_text } from './operations';
import { active_block, untrack } from './runtime';
import { array_from, is_array } from './utils';

function create_item(anchor, value, render_fn) {
 var b = branch(() => {
  render_fn(anchor, value);
 });
 return b;
}

function move(block, anchor) {
 var node = block.s.start;
 var end = block.s.end;

 if (node === end) {
  anchor.before(node);
  return;
 }
 while (node !== end) {
  var next_node = /** @type {TemplateNode} */ (get_next_sibling(node));
  anchor.before(node);
  node = next_node;
 }
}
```

```
export function for_block(node, get_collection, render_fn, flags) {
 var is_controlled = (flags & IS_CONTROLLED) !== 0;
 var anchor = node;

 if (is_controlled) {
  anchor = node.appendChild(create_text());
 }

 render(() => {
  var block = active_block;
  var collection = get_collection();
  var array = is_array(collection)
   ? collection
   : collection == null
   ? []
   : array_from(collection);

  if (array[TRACKED_OBJECT] !== undefined) {
   // TODO we previously assigned array to this, but why?
   get_all_elements(collection);
   collection.$length;
  }

  untrack(() => reconcile(anchor, block, array, render_fn, is_controlled));
 }, FOR_BLOCK);
}

function reconcile_fast_clear(anchor, block, array) {
 var state = block.s;
 var parent_node = anchor.parentNode;
 parent_node.textContent = '';
 destroy_block_children(block);
 parent_node.append(anchor);
 state.array = array;
 state.blocks = [];
}

function reconcile(anchor, block, b, render_fn, is_controlled) {
 var state = block.s;

 if (state === null) {
  state = block.s = {
   array: [],
   blocks: []
  };
 }

 var a = state.array;
 var a_length = a.length;
 var b_length = b.length;
 var j = 0;

 // Fast-path for clear
 if (is_controlled && b_length === 0) {
  if (a_length > 0) {
   reconcile_fast_clear(anchor, block, b);
  }
  return;
 }
 var b_blocks = Array(b_length);
```

```
// Fast-path for create
if (a_length === 0) {
 for (; j < b_length; j++) {
  b_blocks[j] = create_item(anchor, b[j], render_fn);
 }
 state.array = b;
 state.blocks = b_blocks;
 return;
}

var a_blocks = state.blocks;
var a_val = a[j];
var b_val = b[j];
var a_end = a_length - 1;
var b_end = b_length - 1;

outer: {
 while (a_val === b_val) {
  a[j] = b_val;
  b_blocks[j] = a_blocks[j];
  ++j;
  if (j > a_end || j > b_end) {
   break outer;
  }
  a_val = a[j];
  b_val = b[j];
 }

 a_val = a[a_end];
 b_val = b[b_end];

 while (a_val === b_val) {
  a[a_end] = b_val;
  b_blocks[b_end] = a_blocks[a_end];
  a_end--;
  b_end--;
  if (j > a_end || j > b_end) {
   break outer;
  }
  a_val = a[a_end];
  b_val = b[b_end];
 }
}

if (j > a_end) {
 if (j <= b_end) {
  while (j <= b_end) {
   b_val = b[j];
   var target = j >= a_length ? anchor : a_blocks[j].s.start;
   b_blocks[j++] = create_item(target, b_val, render_fn);
  }
 }
} else if (j > b_end) {
 while (j <= a_end) {
  destroy_block(a_blocks[j++]);
 }
} else {
 var a_start = j;
 var b_start = j;
```

```
  var a_left = a_end - j + 1;
  var b_left = b_end - j + 1;
  var fast_path_removal = is_controlled && a_left === a_length;
  var sources = new Int32Array(b_left + 1);
  var moved = false;
  var pos = 0;
  var patched = 0;
  var i = 0;

  // When sizes are small, just loop them through
  if (b_length < 4 || (a_left | b_left) < 32) {
   for (i = a_start; i <= a_end; ++i) {
    a_val = a[i];
    if (patched < b_left) {
     for (j = b_start; j <= b_end; j++) {
      b_val = b[j];
      if (a_val === b_val) {
       sources[j - b_start] = i + 1;
       if (fast_path_removal) {
        fast_path_removal = false;
        // while (a_start < i) {
        //     debugger
        //     destroy_block(a_blocks[a_start++]);
        // }
       }
       if (pos > j) {
        moved = true;
       } else {
        pos = j;
       }
       b_blocks[j] = a_blocks[i];
       ++patched;
       break;
      }
     }
     if (!fast_path_removal && j > b_end) {
      destroy_block(a_blocks[i]);
     }
    } else if (!fast_path_removal) {
     destroy_block(a_blocks[i]);
    }
   }
  } else {
   var map = new Map();

   for (i = b_start; i <= b_end; ++i) {
    map.set(b[i], i);
   }

   for (i = a_start; i <= a_end; ++i) {
    a_val = a[i];

    if (patched < b_left) {
     j = map.get(a_val);

     if (j !== undefined) {
      if (fast_path_removal) {
       fast_path_removal = false;
       // while (i > aStart) {
       //     remove(a[aStart++], dom, animations);
```

```
       // }
      }
      sources[j - b_start] = i + 1;
      if (pos > j) {
       moved = true;
      } else {
       pos = j;
      }
      b_val = b[j];
      b_blocks[j] = a_blocks[i];
      ++patched;
     } else if (!fast_path_removal) {
      destroy_block(a_blocks[i]);
     }
    } else if (!fast_path_removal) {
     destroy_block(a_blocks[i]);
    }
   }
  }
 }

 if (fast_path_removal) {
  reconcile_fast_clear(anchor, block, []);
  reconcile(anchor, block, b, render_fn, is_controlled);
  return;
 } else if (moved) {
  var next_pos = 0;
  var seq = lis_algorithm(sources);
  j = seq.length - 1;

  for (i = b_left - 1; i >= 0; i--) {
   if (sources[i] === 0) {
    pos = i + b_start;
    b_val = b[pos];
    next_pos = pos + 1;

    var target = next_pos < b_length ? b_blocks[next_pos].s.start : anchor;
    b_blocks[pos] = create_item(target, b_val, render_fn);
   } else if (j < 0 || i !== seq[j]) {
    pos = i + b_start;
    b_val = b[pos];
    next_pos = pos + 1;

    var target = next_pos < b_length ? b_blocks[next_pos].s.start : anchor;
    move(b_blocks[pos], target);
   } else {
    j--;
   }
  }
 } else if (patched !== b_left) {
  for (i = b_left - 1; i >= 0; i--) {
   if (sources[i] === 0) {
    pos = i + b_start;
    b_val = b[pos];
    next_pos = pos + 1;

    var target = next_pos < b_length ? b_blocks[next_pos].s.start : anchor;
    b_blocks[pos] = create_item(target, b_val, render_fn);
   }
  }
```

```
 }

 state.array = b;
 state.blocks = b_blocks;
}

let result;
let p;
let maxLen = 0;
// https://en.wikipedia.org/wiki/Longest_increasing_subsequence
function lis_algorithm(arr) {
 let arrI = 0;
 let i = 0;
 let j = 0;
 let k = 0;
 let u = 0;
 let v = 0;
 let c = 0;
 var len = arr.length;

 if (len > maxLen) {
  maxLen = len;
  result = new Int32Array(len);
  p = new Int32Array(len);
 }

 for (; i < len; ++i) {
  arrI = arr[i];

  if (arrI !== 0) {
   j = result[k];
   if (arr[j] < arrI) {
    p[i] = j;
    result[++k] = i;
    continue;
   }

   u = 0;
   v = k;

   while (u < v) {
    c = (u + v) >> 1;
    if (arr[result[c]] < arrI) {
     u = c + 1;
    } else {
     v = c;
    }
   }

   if (arrI < arr[result[u]]) {
    if (u > 0) {
     p[i] = result[u - 1];
    }
    result[u] = i;
   }
  }
 }

 u = k + 1;
 var seq = new Int32Array(u);
```

```
  v = result[u - 1];

  while (u-- > 0) {
   seq[u] = v;
   v = p[v];
   result[u] = 0;
  }

  return seq;
}

export function keyed(collection, key_fn) {
  var block = active_block;
  if (block === null || (block.f & FOR_BLOCK) === 0) {
   throw new Error('keyed() must be used inside a for block');
  }
  var state = block.s;

  if (state === null) {
   return collection;
  }

  var a_array = state.array;
  var a_keys = a_array.map(key_fn);
  var a = new Map();

  for (let i = 0; i < a_keys.length; i++) {
   a.set(a_keys[i], i);
  }

  if (a.size !== a_keys.length) {
   throw new Error('Duplicate keys are not allowed');
  }

  var b_array = is_array(collection)
   ? collection
   : collection == null
   ? []
   : array_from(collection);
  var b_keys = b_array.map(key_fn);

  // We only need to do this in DEV
  var b = new Set(b_keys);
  if (b.size !== b_keys.length) {
   throw new Error('Duplicate keys are not allowed');
  }

  for (let i = 0; i < b_keys.length; i++) {
   var b_val = b_keys[i];
   var index = a.get(b_val);

   if (index !== undefined) {
    b_array[i] = a_array[index];
   }
  }

  return b_array;
}
```

src/runtime/internal/client/if.js

```
import { branch, destroy_block, render } from './blocks';
import { IF_BLOCK, UNINITIALIZED } from './constants';

export function if_block(node, fn) {
 var anchor = node;
 var has_branch = false;
 var condition = UNINITIALIZED;
 var b = null;

 var set_branch = (fn, flag = true) => {
  has_branch = true;
  update_branch(flag, fn);
 };

 var update_branch = (new_condition, fn) => {
  if (condition === (condition = new_condition)) return;

  if (b !== null) {
   destroy_block(b);
   b = null;
  }

  if (fn !== null) {
   b = branch(() => fn(anchor));
  }
 };

 render(() => {
  has_branch = false;
  fn(set_branch);
  if (!has_branch) {
   update_branch(null, null);
  }
 }, IF_BLOCK);
}
```

src/runtime/internal/client/index.js

```
export { first_child as child, child_frag, next_sibling as sibling } from './operations.js';

export {
 set_text,
 set_class,
 set_attribute,
 set_value,
 set_checked,
 set_selected,
 set_ref
} from './render.js';

export { render, async } from './blocks.js';

export { event, delegate } from './events.js';
```

```
export {
 active_block,
 scope,
 with_scope,
 get_tracked,
 get_computed,
 set,
 computed,
 async_computed,
 tracked,
 tracked_object,
 computed_property,
 get_property,
 set_property,
 update,
 update_pre,
 update_property,
 update_pre_property,
 object_values,
 object_entries,
 object_keys,
 spread_object,
 structured_clone,
 push_component,
 pop_component,
 untrack
} from './runtime.js';

export { for_block as for } from './for.js';

export { if_block as if } from './if.js';

export { try_block as try, resume_context, aborted } from './try.js';

export { template, append } from './template.js';
```

src/runtime/internal/client/operations.js

```
import { TRACKED_OBJECT } from './constants.js';
import { get_descriptor } from './utils.js';

/** @type {() => Node | null} */
var first_child_getter;
/** @type {() => Node | null} */
var next_sibling_getter;

export var is_firefox;

export function init_operations() {
 var node_prototype = Node.prototype;
 var element_prototype = Element.prototype;
 var object_prototype = Object.prototype;
 var event_target_prototype = EventTarget.prototype;

 is_firefox = /Firefox/.test(navigator.userAgent);

 // @ts-ignore
```

```
 first_child_getter = get_descriptor(node_prototype, 'firstChild').get;
 // @ts-ignore
 next_sibling_getter = get_descriptor(node_prototype, 'nextSibling').get;

 // the following assignments improve perf of lookups on DOM nodes
 // @ts-expect-error
 element_prototype.__click = undefined;
 // @ts-expect-error
 element_prototype.__className = '';
 // @ts-expect-error
 element_prototype.__attributes = null;
 // @ts-expect-error
 element_prototype.__styles = null;
 // @ts-expect-error
 element_prototype.__e = undefined;
 // @ts-expect-error
 object_prototype[TRACKED_OBJECT] = undefined;
 // @ts-expect-error
 event_target_prototype.__root = undefined;
}

/**
 * @template {Node} N
 * @param {N} node
 * @returns {Node | null}
 */
/*@__NO_SIDE_EFFECTS__*/
export function first_child(node) {
 return first_child_getter.call(node);
}

export function child_frag(node)  {
 var child = first_child(node);

 if (child.nodeType === 8 && child.data === '') {
  return next_sibling(child);
 }
 return child;
}

/**
 * @template {Node} N
 * @param {N} node
 * @returns {Node | null}
 */
/*@__NO_SIDE_EFFECTS__*/
export function next_sibling(node) {
 return next_sibling_getter.call(node);
}

export function create_text(value = '') {
 return document.createTextNode(value);
}
```

src/runtime/internal/client/portal.js

```
import { branch, destroy_block, render } from './blocks';
import { UNINITIALIZED } from './constants';
```

```
import { handle_root_events } from './events';
import { create_text } from './operations';
import { get_property } from './runtime';

export function Portal(_, props) {
 let $target = UNINITIALIZED;
 let children = UNINITIALIZED;
 var b = null;
 var anchor = null;

 render(() => {
  if ($target === ($target = get_property(props, '$target'))) return;
  if (children === (children = get_property(props, 'children'))) return;

  if (b !== null) {
   destroy_block(b);
  }

  anchor = create_text();
  $target.append(anchor);

  const cleanup_events = handle_root_events($target);

  b = branch(() => children(anchor));

  return () => {
   cleanup_events();
   anchor.remove();
  };
 });
}
```

src/runtime/internal/client/render.js

```
import { effect } from './blocks';
import { get_descriptors, get_prototype_of } from './utils';

export function set_text(text, value) {
 // For objects, we apply string coercion (which might make things like $state array references in t
he template reactive) before diffing
 var str = value == null ? '' : typeof value === 'object' ? value + '' : value;
 // @ts-expect-error
 if (str !== (text.__t ??= text.nodeValue)) {
  // @ts-expect-error
  text.__t = str;
  text.nodeValue = str + '';
 }
}

var setters_cache = new Map();

function get_setters(element) {
 var setters = setters_cache.get(element.nodeName);
 if (setters) return setters;
 setters_cache.set(element.nodeName, (setters = []));

 var descriptors;
 var proto = element; // In the case of custom elements there might be setters on the instance
```

```
 var element_proto = Element.prototype;

 // Stop at Element, from there on there's only unnecessary setters we're not interested in
 // Do not use contructor.name here as that's unreliable in some browser environments
 while (element_proto !== proto) {
  descriptors = get_descriptors(proto);

  for (var key in descriptors) {
   if (descriptors[key].set) {
    setters.push(key);
   }
  }

  proto = get_prototype_of(proto);
 }

 return setters;
}

export function set_attribute(element, attribute, value) {
 // @ts-expect-error
 var attributes = (element.__attributes ??= {});

 if (attributes[attribute] === (attributes[attribute] = value)) return;

 if (attribute === 'style' && '__styles' in element) {
  // reset styles to force style: directive to update
  element.__styles = {};
 }

 if (value == null) {
  element.removeAttribute(attribute);
 } else if (typeof value !== 'string' && get_setters(element).includes(attribute)) {
  element[attribute] = value;
 } else {
  element.setAttribute(attribute, value);
 }
}

/**
 * @template V
 * @param {V} value
 * @param {string} [hash]
 * @returns {string | V}
 */
function to_class(value, hash) {
 return (value == null ? '' : value) + (hash ? ' ' + hash : '');
}

/**
 * @param {HTMLElement} dom
 * @param {string} value
 * @param {string} [hash]
 * @returns {void}
 */
export function set_class(dom, value, hash) {
 // @ts-expect-error need to add __className to patched prototype
 var prev_class_name = dom.__className;
 var next_class_name = to_class(value, hash);
```

```
  if (prev_class_name !== next_class_name) {
   // Removing the attribute when the value is only an empty string causes
   // peformance issues vs simply making the className an empty string. So
   // we should only remove the class if the the value is nullish.
   if (value == null && !hash) {
    dom.removeAttribute('class');
   } else {
    dom.className = next_class_name;
   }

   // @ts-expect-error need to add __className to patched prototype
   dom.__className = next_class_name;
  }
 }

 export function set_value(element, value) {
  // @ts-expect-error
  var attributes = (element.__attributes ??= {});

  if (
   attributes.value ===
    (attributes.value =
     // treat null and undefined the same for the initial value
     value ?? undefined) ||
   // @ts-expect-error
   // `progress` elements always need their value set when it's `0`
   (element.value === value && (value !== 0 || element.nodeName !== 'PROGRESS'))
  ) {
   return;
  }

  // @ts-expect-error
  element.value = value ?? '';
 }

 export function set_checked(element, checked) {
  // @ts-expect-error
  var attributes = (element.__attributes ??= {});

  if (
   attributes.checked ===
   (attributes.checked =
    // treat null and undefined the same for the initial value
    checked ?? undefined)
  ) {
   return;
  }

  // @ts-expect-error
  element.checked = checked;
 }

 export function set_selected(element, selected) {
  if (selected) {
   // The selected option could've changed via user selection, and
   // setting the value without this check would set it back.
   if (!element.hasAttribute('selected')) {
    element.setAttribute('selected', '');
   }
  } else {
```

```
  element.removeAttribute('selected');
 }
}

export function set_ref(dom, fn) {
 effect(() => {
  fn(dom);

  return () => {
   fn(null);
  };
 });
}
```

src/runtime/internal/client/runtime.js

```
import {
 destroy_block,
 destroy_non_branch_children,
 effect,
 is_destroyed,
 render
} from './blocks.js';
import {
 ASYNC_BLOCK,
 BLOCK_HAS_RUN,
 BRANCH_BLOCK,
 COMPUTED,
 COMPUTED_PROPERTY,
 CONTAINS_TEARDOWN,
 CONTAINS_UPDATE,
 DEFERRED,
 DESTROYED,
 EFFECT_BLOCK,
 PAUSED,
 ROOT_BLOCK,
 TRACKED,
 TRACKED_OBJECT,
 TRY_BLOCK,
 UNINITIALIZED
} from './constants';
import { capture, suspend } from './try.js';
import { define_property, is_array } from './utils';
import {
 object_keys as original_object_keys,
 object_values as original_object_values,
 object_entries as original_object_entries,
 structured_clone as original_structured_clone
} from './utils.js';

const FLUSH_MICROTASK = 0;
const FLUSH_SYNC = 1;

export let active_block = null;
export let active_reaction = null;
export let active_scope = null;
export let active_component = null;
```

```
var old_values = new Map();

// Used for controlling the flush of blocks
let scheduler_mode = FLUSH_MICROTASK;
// Used for handling scheduling
let is_micro_task_queued = false;
let clock = 0;
let queued_root_blocks = [];
let queued_microtasks = [];
let flush_count = 0;
let active_dependency = null;

export let tracking = false;
export let teardown = false;

function increment_clock() {
 return ++clock;
}

export function set_active_block(block) {
 active_block = block;
}

export function set_active_reaction(reaction) {
 active_reaction = reaction;
}

export function set_active_component(component) {
 active_component = component;
}

export function set_tracking(value) {
 tracking = value;
}

export function run_teardown(block) {
 var fn = block.t;
 if (fn !== null) {
  var previous_block = active_block;
  var previous_reaction = active_reaction;
  var previous_tracking = tracking;
  var previous_teardown = teardown;

  try {
   active_block = null;
   active_reaction = null;
   tracking = false;
   teardown = true;
   fn.call(null);
  } finally {
   active_block = previous_block;
   active_reaction = previous_reaction;
   tracking = previous_tracking;
   teardown = previous_teardown;
  }
 }
}

function update_computed(computed) {
 var value = computed.v;
```

```
 if (value === UNINITIALIZED || is_tracking_dirty(computed.d)) {
  value = run_computed(computed);

  if (value !== computed.v) {
   computed.v = value;
   computed.c = increment_clock();
  }
 }
}

function destroy_computed_children(computed) {
 var blocks = computed.blocks;

 if (blocks !== null) {
  computed.blocks = null;
  for (var i = 0; i < blocks.length; i++) {
   destroy_block(blocks[i]);
  }
 }
}

function run_computed(computed) {
 var previous_block = active_block;
 var previous_reaction = active_reaction;
 var previous_tracking = tracking;
 var previous_dependency = active_dependency;
 var previous_component = active_component;

 try {
  active_block = computed.b;
  active_reaction = computed;
  tracking = true;
  active_dependency = null;
  active_component = active_block.c;

  destroy_computed_children(computed);

  var value = computed.fn();

  computed.d = active_dependency;

  return value;
 } finally {
  active_block = previous_block;
  active_reaction = previous_reaction;
  tracking = previous_tracking;
  active_dependency = previous_dependency;
  active_component = previous_component;
 }
}

export function handle_error(error, block) {
 var current = block;

 while (current !== null) {
  var state = current.s;
  if ((current.f & TRY_BLOCK) !== 0 && state.c !== null) {
   state.c(error);
   return;
```

```
   }
   current = current.p;
  }

  throw error;
 }

 export function run_block(block) {
  var previous_block = active_block;
  var previous_reaction = active_reaction;
  var previous_tracking = tracking;
  var previous_dependency = active_dependency;
  var previous_component = active_component;

  try {
   active_block = block;
   active_reaction = block;
   active_component = block.c;

   destroy_non_branch_children(block);
   run_teardown(block);

   tracking = (block.f & (ROOT_BLOCK | BRANCH_BLOCK)) === 0;
   active_dependency = null;
   var res = block.fn();

   if (typeof res === 'function') {
    block.t = res;
    let current = block;

    while (current !== null && (current.f & CONTAINS_TEARDOWN) === 0) {
     current.f ^= CONTAINS_TEARDOWN;
     current = current.p;
    }
   }

   block.d = active_dependency;
  } catch (error) {
   handle_error(error, block);
  } finally {
   active_block = previous_block;
   active_reaction = previous_reaction;
   tracking = previous_tracking;
   active_dependency = previous_dependency;
   active_component = previous_component;
  }
 }

 export function tracked(v, b) {
  return {
   b,
   c: 0,
   f: TRACKED,
   v
  };
 }

 export function computed(fn, b) {
  return {
   b,
```

```
   blocks: null,
   c: 0,
   d: null,
   f: TRACKED | COMPUTED,
   fn,
   v: UNINITIALIZED
  };
}

function create_dependency(tracked) {
 var existing = active_reaction.d;

 // Recycle tracking entries
 if (existing !== null) {
  active_reaction.d = existing.n;
  existing.c = tracked.c;
  existing.t = tracked;
  existing.n = null;
  return existing;
 }

 return {
  c: tracked.c,
  t: tracked,
  n: null
 };
}

function is_tracking_dirty(tracking) {
 if (tracking === null) {
  return false;
 }
 while (tracking !== null) {
  var tracked = tracking.t;

  if ((tracked.f & COMPUTED) !== 0) {
   update_computed(tracked);
  }

  if (tracked.c > tracking.c) {
   return true;
  }
  tracking = tracking.n;
 }

 return false;
}

function is_block_dirty(block) {
 var flags = block.f;

 if ((flags & (ROOT_BLOCK | BRANCH_BLOCK)) !== 0) {
  return false;
 }
 if ((flags & BLOCK_HAS_RUN) === 0) {
  block.f ^= BLOCK_HAS_RUN;
  return true;
 }

 return is_tracking_dirty(block.d);
```

```
}

export function async_computed(fn, block) {
 let parent = active_reaction;
 var t = tracked(UNINITIALIZED, block);
 var promise;
 var new_values = new Map();

 render(() => {
  var [current, deferred] = capture_deferred(() => (promise = fn()));

  var restore = capture();
  var unuspend;

  if (deferred === null) {
   unuspend = suspend();
  } else {
   for (var i = 0; i < deferred.length; i++) {
    var tracked = deferred[i];
    new_values.set(tracked, { v: tracked.v, c: tracked.c });
   }
  }

  promise.then((v) => {
   if (is_destroyed(parent)) {
    return;
   }
   if (promise === current && t.v !== v) {
    restore();

    if (t.v === UNINITIALIZED) {
     t.v = v;
    } else {
     set(t, v, block);
    }
   }

   if (deferred === null) {
    unuspend();
   } else if (promise === current) {
    for (var i = 0; i < deferred.length; i++) {
     var tracked = deferred[i];
     var { v, c } = new_values.get(tracked);
     tracked.v = v;
     tracked.c = c;
     schedule_update(tracked.b);
    }
    new_values.clear();
   }
  });
 }, ASYNC_BLOCK);

 return new Promise(async (resolve) => {
  var p;
  while (p !== (p = promise)) await p;
  return resolve(t);
 });
}

export function deferred(fn) {
```

```
 var parent = active_block;
 var block = active_scope;
 var res = [UNINITIALIZED];
 var t = tracked(UNINITIALIZED, block, DEFERRED);
 var tracked_properties = [t];
 var prev_value = UNINITIALIZED;

 define_property(res, TRACKED_OBJECT, {
  value: tracked_properties,
  enumerable: false
 });

 render(() => {
  if (prev_value !== UNINITIALIZED) {
   t.v = prev_value;
  } else {
   prev_value = t.v;
  }
  var prev_version = t.c;
  var value = fn();

  res[0] = value;
  set_property(res, 0, value, block);

  if (prev_value !== UNINITIALIZED) {
   if ((t.f & DEFERRED) === 0) {
    t.f ^= DEFERRED;
   }

   var is_awaited = flush_deferred_upodates(parent);
   if ((t.f & DEFERRED) !== 0) {
    t.f ^= DEFERRED;
   }

   if (is_awaited) {
    t.c = prev_version;
    t.v = prev_value;
    prev_value = value;
   }
  }
 });

 return res;
}

function capture_deferred(fn) {
 var value = fn();
 var deferred = null;
 var depedency = active_dependency;

 while (depedency !== null) {
  var tracked = depedency.t;
  if ((tracked.f & DEFERRED) !== 0) {
   deferred ??= [];
   deferred.push(tracked);
   break;
  }
  depedency = depedency.n;
 }
```

```
 return [value, deferred];
}

function flush_deferred_upodates(block) {
 var current = block.first;
 var is_awaited = false;

 main_loop: while (current !== null) {
  var flags = current.f;

  if ((flags & ASYNC_BLOCK) !== 0 && is_block_dirty(current)) {
   is_awaited = true;
   run_block(current);
  }

  var parent = current.p;
  current = current.next;

  while (current === null && parent !== null) {
   if (parent === block) {
    break main_loop;
   }
   current = parent.next;
   parent = parent.p;
  }
 }

 return is_awaited;
}

function flush_updates(root_block) {
 var current = root_block;
 var containing_update = null;
 var effects = [];

 while (current !== null) {
  var flags = current.f;

  if ((flags & CONTAINS_UPDATE) !== 0) {
   current.f ^= CONTAINS_UPDATE;
   containing_update = current;
  }

  if ((flags & PAUSED) === 0 && containing_update !== null) {
   if ((flags & EFFECT_BLOCK) !== 0) {
    effects.push(current);
   } else {
    try {
     if (is_block_dirty(current)) {
      run_block(current);
     }
    } catch (error) {
     handle_error(error, current);
    }
   }
   var child = current.first;

   if (child !== null) {
    current = child;
    continue;
```

```
    }
   }

   var parent = current.p;
   current = current.next;

   while (current === null && parent !== null) {
    if (parent === containing_update) {
     containing_update = null;
    }
    current = parent.next;
    parent = parent.p;
   }
  }

  var length = effects.length;

  for (var i = 0; i < length; i++) {
   var effect = effects[i];
   var flags = effect.f;

   try {
    if ((flags & (PAUSED | DESTROYED)) === 0 && is_block_dirty(effect)) {
     run_block(effect);
    }
   } catch (error) {
    handle_error(error, effect);
   }
  }
}

function flush_queued_root_blocks(root_blocks) {
 for (let i = 0; i < root_blocks.length; i++) {
  flush_updates(root_blocks[i]);
 }
}

function flush_microtasks() {
 is_micro_task_queued = false;

 if (queued_microtasks.length > 0) {
  var microtasks = queued_microtasks;
  queued_microtasks = [];
  for (var i = 0; i < microtasks.length; i++) {
   microtasks[i]();
  }
 }

 if (flush_count > 1001) {
  return;
 }
 var previous_queued_root_blocks = queued_root_blocks;
 queued_root_blocks = [];
 flush_queued_root_blocks(previous_queued_root_blocks);

 if (!is_micro_task_queued) {
  flush_count = 0;
 }
 old_values.clear();
}
```

```
export function queue_microtask(fn) {
 if (!is_micro_task_queued) {
  is_micro_task_queued = true;
  queueMicrotask(flush_microtasks);
 }
 if (fn !== undefined) {
  queued_microtasks.push(fn);
 }
}

export function schedule_update(block) {
 if (scheduler_mode === FLUSH_MICROTASK) {
  queue_microtask();
 }
 let current = block;

 while (current !== null) {
  var flags = current.f;
  if ((flags & CONTAINS_UPDATE) !== 0) return;
  current.f ^= CONTAINS_UPDATE;
  if ((flags & ROOT_BLOCK) !== 0) {
   break;
  }
  current = current.p;
 }

 queued_root_blocks.push(current);
}

function register_dependency(tracked) {
 var depedency = active_dependency;

 if (depedency === null) {
  depedency = create_dependency(tracked);
  active_dependency = depedency;
 } else {
  var current = depedency;

  while (current !== null) {
   if (current.t === tracked) {
    current.c = tracked.c;
    return;
   }
   var next = current.n;
   if (next === null) {
    break;
   }
   current = next;
  }

  depedency = create_dependency(tracked);
  current.n = depedency;
 }
}

export function get_computed(computed) {
 update_computed(computed);
 if (tracking) {
  register_dependency(computed);
```

```
  }

  return computed.v;
}

export function get(tracked) {
  return (tracked.f & COMPUTED) !== 0 ? get_computed(tracked) : get_tracked(tracked);
}

export function get_tracked(tracked) {
  var value = tracked.v;
  if (tracking) {
    register_dependency(tracked);
  }
  if (teardown && old_values.has(tracked)) {
    return old_values.get(tracked);
  }
  return value;
}

export function set(tracked, value, block) {
  var old_value = tracked.v;

  if (value !== old_value) {
    var tracked_block = tracked.b;

    if ((block.f & CONTAINS_TEARDOWN) !== 0) {
      if (teardown) {
        old_values.set(tracked, value);
      } else {
        old_values.set(tracked, old_value);
      }
    }

    tracked.v = value;
    tracked.c = increment_clock();

    if (tracked_block !== block) {
      throw new Error(
        'Tracked state can only be updated within the same component context that it was created in (tha
t includes effects or event handler within that component).'
      );
    }
    schedule_update(tracked_block);
  }
}

export function untrack(fn) {
  var previous_tracking = tracking;
  var previous_dependency = active_dependency;
  tracking = false;
  active_dependency = null;
  try {
    return fn();
  } finally {
    tracking = previous_tracking;
    active_dependency = previous_dependency;
  }
}
```

```
export function flush_sync(fn) {
 var previous_scheduler_mode = scheduler_mode;
 var previous_queued_root_blocks = queued_root_blocks;

 try {
  const root_blocks = [];

  scheduler_mode = FLUSH_SYNC;
  queued_root_blocks = root_blocks;
  is_micro_task_queued = false;

  flush_queued_root_blocks(previous_queued_root_blocks);

  var result = fn?.();

  if (queued_root_blocks.length > 0 || root_blocks.length > 0) {
   flush_sync();
  }

  flush_count = 0;

  return result;
 } finally {
  scheduler_mode = previous_scheduler_mode;
  queued_root_blocks = previous_queued_root_blocks;
 }
}

export function tracked_object(obj, properties, block) {
 var tracked_properties = obj[TRACKED_OBJECT];

 if (tracked_properties === undefined) {
  tracked_properties = {};
  define_property(obj, TRACKED_OBJECT, {
   value: tracked_properties,
   enumerable: false
  });
 }

 for (var i = 0; i < properties.length; i++) {
  var property = properties[i];
  var initial = obj[property];
  var tracked_property;

  if (typeof initial === 'function' && initial[COMPUTED_PROPERTY] === true) {
   tracked_property = computed(initial, block);
   initial = run_computed(tracked_property);
   obj[property] = initial;
   // TODO If nothing is tracked in the computed function, we can make it a standard tracked
   // however this is more allocations, so we probably want to minimize this
   // if (tracked_property.d === null) {
   //  tracked_property = tracked(initial, block);
   // }
  } else {
   tracked_property = tracked(initial, block);
  }
  tracked_properties[property] = tracked_property;
 }

 return obj;
```

```
}

export function computed_property(fn) {
 define_property(fn, COMPUTED_PROPERTY, {
  value: true,
  enumerable: false
 });
 return fn;
}

export function get_property(obj, property, chain = false) {
 if (chain && obj == null) {
  return undefined;
 }
 var value = obj[property];
 var tracked_properties = obj[TRACKED_OBJECT];
 var tracked_property = tracked_properties?.[property];

 if (tracked_property !== undefined) {
  value = obj[property] = get(tracked_property);
 }

 return value;
}

export function set_property(obj, property, value, block) {
 var res = (obj[property] = value);
 var tracked_properties = obj[TRACKED_OBJECT];
 var tracked = tracked_properties?.[property];

 if (tracked === undefined) {
  return res;
 }

 set(tracked, value, block);
}

export function update(tracked, block, d = 1) {
 var value = get(tracked);
 var result = d === 1 ? value++ : value--;

 set(tracked, value, block);

 return result;
}

export function increment(tracked, block) {
 set(tracked, tracked.v + 1, block);
}

export function update_pre(tracked, block, d = 1) {
 var value = get(tracked);

 return set(tracked, d === 1 ? ++value : --value, block);
}

export function update_property(obj, property, block, d = 1) {
 var tracked_properties = obj[TRACKED_OBJECT];
 var tracked = tracked_properties?.[property];
```

```
 if (tracked === undefined) {
  return d === 1 ? obj[property]++ : obj[property]--;
 }

 var value = get(tracked);
 var result = d === 1 ? value++ : value--;

 increment(tracked, block);
 return result;
}

export function update_pre_property(obj, property, block, d = 1) {
 var tracked_properties = obj[TRACKED_OBJECT];
 var tracked = tracked_properties?.[property];

 if (tracked === undefined) {
  return d === 1 ? ++obj[property] : --obj[property];
 }

 var value = get(tracked);
 var result = d === 1 ? ++value : --value;

 increment(tracked, block);
 return result;
}

export function structured_clone(val, options) {
 if (typeof val === 'object' && val !== null) {
  var tracked_properties = val[TRACKED_OBJECT];
  if (tracked_properties !== undefined) {
   if (is_array(val)) {
    val.$length;
   }
   return structured_clone(object_values(val), options);
  }
 }
 return original_structured_clone(val, options);
}

export function object_keys(obj) {
 if (is_array(obj) && TRACKED_OBJECT in obj) {
  obj.$length;
 }
 return original_object_keys(obj);
}

export function object_values(obj) {
 var tracked_properties = obj[TRACKED_OBJECT];

 if (tracked_properties === undefined) {
  return original_object_values(obj);
 }
 if (is_array(obj)) {
  obj.$length;
 }
 var keys = original_object_keys(obj);
 var values = [];

 for (var i = 0; i < keys.length; i++) {
  values.push(get_property(obj, keys[i]));
```

```
  }

  return values;
}

export function object_entries(obj) {
  var tracked_properties = obj[TRACKED_OBJECT];

  if (tracked_properties === undefined) {
    return original_object_entries(obj);
  }
  if (is_array(obj)) {
    obj.$length;
  }
  var keys = original_object_keys(obj);
  var entries = [];

  for (var i = 0; i < keys.length; i++) {
    var key = keys[i];
    entries.push([key, get_property(obj, key)]);
  }

  return entries;
}

export function spread_object(obj) {
  var tracked_properties = obj[TRACKED_OBJECT];

  if (tracked_properties === undefined) {
    return { ...obj };
  }
  var keys = original_object_keys(obj);
  const values = {};

  for (var i = 0; i < keys.length; i++) {
    var key = keys[i];
    values[key] = get_property_computed(obj, key);
  }

  return values;
}

export function with_scope(block, fn) {
  var previous_scope = active_scope;
  try {
    active_scope = block;
    return fn();
  } finally {
    active_scope = previous_scope;
  }
}

export function scope() {
  return active_scope;
}

export function push_component() {
  var component = {
    e: null,
    m: false,
```

```
  p: active_component
 };
 active_component = component;
}

export function pop_component() {
 var component = active_component;
 component.m = true;
 var effects = component.e;
 if (effects !== null) {
  var length = effects.length;
  for (var i = 0; i < length; i++) {
   var { b: block, fn, r: reaction } = effects[i];
   var previous_block = active_block;
   var previous_reaction = active_reaction;

   try {
    active_block = block;
    active_reaction = reaction;
    effect(fn);
   } finally {
    active_block = previous_block;
    active_reaction = previous_reaction;
   }
  }
 }
 active_component = component.p;
}
```

src/runtime/internal/client/template.js

```
import { TEMPLATE_FRAGMENT, TEMPLATE_USE_IMPORT_NODE } from '../../../constants.js';
import { first_child, is_firefox } from './operations.js';
import { active_block } from './runtime.js';

export function assign_nodes(start, end) {
 var block = /** @type {Effect} */ (active_block);
 if (block.s === null) {
  block.s = {
   start,
   end
  };
 }
}

function create_fragment_from_html(html) {
 var elem = document.createElement('template');
 elem.innerHTML = html;
 return elem.content;
}

export function template(content, flags) {
 var is_fragment = (flags & TEMPLATE_FRAGMENT) !== 0;
 var use_import_node = (flags & TEMPLATE_USE_IMPORT_NODE) !== 0;
 var node;
 var has_start = !content.startsWith('<!>');

 return () => {
```

```
  if (node === undefined) {
   node = create_fragment_from_html(has_start ? content : '<!>' + content);
   if (!is_fragment) node = first_child(node);
  }

  var clone =
   use_import_node || is_firefox ? document.importNode(node, true) : node.cloneNode(true);

  if (is_fragment) {
   var start = first_child(clone);
   var end = clone.lastChild;

   assign_nodes(start, end);
  } else {
   assign_nodes(clone, clone);
  }

  return clone;
 };
}

export function append(anchor, dom) {
 anchor.before(/** @type {Node} */ (dom));
}
```

src/runtime/internal/client/try.js

```
import { branch, create_try_block, destroy_block, is_destroyed, resume_block } from './blocks';
import { TRY_BLOCK } from './constants';
import { next_sibling } from './operations';
import {
 active_block,
 active_component,
 active_reaction,
 queue_microtask,
 set_active_block,
 set_active_component,
 set_active_reaction,
 set_tracking,
 tracking
} from './runtime';

export function try_block(node, fn, catch_fn, pending_fn = null) {
 var anchor = node;
 var b = null;
 var suspended = null;
 var pending_count = 0;
 var offscreen_fragment = null;

 function move_block(block, fragment) {
  var state = block.s;
  var node = state.start;
  var end = state.end;

  while (node !== null) {
   var next = node === end ? null : next_sibling(node);

   fragment.append(node);
```

```
    node = next;
   }
  }

  function handle_await() {
   if (pending_count++ === 0) {
    queue_microtask(() => {
     if (b !== null) {
      suspended = b;
      offscreen_fragment = document.createDocumentFragment();
      move_block(b, offscreen_fragment);

      b = branch(() => {
       pending_fn(anchor);
      });
     }
    });
   }

   return () => {
    if (--pending_count === 0) {
     if (b !== null) {
      destroy_block(b);
     }
     anchor.before(offscreen_fragment);
     offscreen_fragment = null;
     resume_block(suspended);
     b = suspended;
     suspended = null;
    }
   };
  }

  function handle_error(error) {
   if (b !== null) {
    destroy_block(b);
   }

   b = branch(() => {
    catch_fn(anchor, error);
   });
  }

  var state = {
   a: pending_fn !== null ? handle_await : null,
   c: catch_fn !== null ? handle_error : null
  };

  create_try_block(() => {
   b = branch(() => {
    fn(anchor);
   });
  }, state);
 }

 export function suspend() {
  var current = active_block;

  while (current !== null) {
   var state = current.s;
```

```
  if ((current.f & TRY_BLOCK) !== 0 && state.a !== null) {
   return state.a();
  }
  current = current.p;
 }

 throw new Error('Missing parent `try { ... } async { ... }` statement');
}

function exit() {
 set_tracking(false);
 set_active_reaction(null);
 set_active_block(null);
 set_active_component(null);
}

export function capture() {
 var previous_tracking = tracking;
 var previous_block = active_block;
 var previous_reaction = active_reaction;
 var previous_component = active_component;

 return () => {
  set_tracking(previous_tracking);
  set_active_block(previous_block);
  set_active_reaction(previous_reaction);
  set_active_component(previous_component);

  queue_microtask(exit);
 };
}

export function aborted() {
 if (active_block === null) {
  return true;
 }
 return is_destroyed(active_block);
}

export async function resume_context(promise) {
 var restore = capture();
 var value = await promise;

 return () => {
  restore();
  return value;
 };
}
```

src/runtime/internal/client/utils.js

```
export var get_descriptor = Object.getOwnPropertyDescriptor;
export var get_descriptors = Object.getOwnPropertyDescriptors;
export var array_from = Array.from;
export var is_array = Array.isArray;
export var define_property = Object.defineProperty;
export var get_prototype_of = Object.getPrototypeOf;
export var object_values = Object.values;
```

```
export var object_entries = Object.entries;
export var object_keys = Object.keys;
export var structured_clone = structuredClone;

export function create_anchor() {
 var t = document.createTextNode('');
 t.__t = '';
 return t;
}
```

src/runtime/array.js

```
import { TRACKED_OBJECT } from './internal/client/constants.js';
import { get, increment, scope, set, tracked } from './internal/client/runtime.js';

var symbol_iterator = Symbol.iterator;

const introspect_methods = [
 'entries',
 'every',
 'find',
 'findIndex',
 'findLast',
 'findLastIndex',
 'flat',
 'flatMap',
 'forEach',
 'includes',
 'indexOf',
 'join',
 'keys',
 'lastIndexOf',
 'map',
 'reduce',
 'reduceRight',
 'some',
 'slice',
 'toLocaleString',
 'toReversed',
 'toSorted',
 'toSpliced',
 'toString',
 symbol_iterator,
 'values',
 'with'
];

let init = false;

class RippleArray extends Array {
 #tracked_elements = [];
 #tracked_index;

 constructor(...elements) {
  super(...elements);

  var block = scope();
  var tracked_elements = this.#tracked_elements;
```

```
  for (var i = 0; i < this.length; i++) {
   tracked_elements[i] = tracked(0, block);
  }
  this.#tracked_index = tracked(this.length, block);

  if (!init) {
   init = true;
   this.#init();
  }
 }

 #init() {
  var proto = RippleArray.prototype;
  var array_proto = Array.prototype;

  for (const method of introspect_methods) {
   proto[method] = function (...v) {
    this.$length;
    get_all_elements(this);
    return array_proto[method].apply(this, v);
   };
  }
 }

 fill() {
  var block = scope();
  var tracked_elements = this.#tracked_elements;

  super.fill();
  for (var i = 0; i < this.length; i++) {
   increment(tracked_elements[i], block);
  }
 }

 reverse() {
  var block = scope();
  var tracked_elements = this.#tracked_elements;

  super.reverse();
  for (var i = 0; i < this.length; i++) {
   increment(tracked_elements[i], block);
  }
 }

 sort(fn) {
  var block = scope();
  var tracked_elements = this.#tracked_elements;

  super.sort(fn);
  for (var i = 0; i < this.length; i++) {
   increment(tracked_elements[i], block);
  }
 }

 unshift(...elements) {
  var block = scope();
  var tracked_elements = this.#tracked_elements;

  super.unshift(...elements);
```

```
  for (var i = 0; i < tracked_elements.length; i++) {
   increment(tracked_elements[i], block);
  }
  tracked_elements.unshift(...elements.map(() => tracked(0, block)));

  set(this.#tracked_index, this.length, block);
 }

 shift() {
  var block = scope();
  var tracked_elements = this.#tracked_elements;

  super.shift();
  for (var i = 0; i < tracked_elements.length; i++) {
   increment(tracked_elements[i], block);
  }
  tracked_elements.shift();

  set(this.#tracked_index, this.length, block);
 }

 push(...elements) {
  var block = scope();
  var start_index = this.length;
  var tracked_elements = this.#tracked_elements;

  super.push(...elements);

  for (var i = 0; i < elements.length; i++) {
   tracked_elements[start_index + i] = tracked(0, block);
  }
  set(this.#tracked_index, this.length, block);
 }

 pop() {
  var block = scope();
  var tracked_elements = this.#tracked_elements;

  super.pop();
  if (tracked_elements.length > 0) {
   increment(tracked_elements[tracked_elements.length - 1], block);
  }
  tracked_elements.pop();

  set(this.#tracked_index, this.length, block);
 }

 splice(start, delete_count, ...elements) {
  var block = scope();
  var tracked_elements = this.#tracked_elements;

  super.splice(start, delete_count, ...elements);
  for (var i = 0; i < tracked_elements.length; i++) {
   increment(tracked_elements[i], block);
  }
  tracked_elements.splice(start, delete_count, ...elements.map(() => tracked(0, block)));

  set(this.#tracked_index, this.length, block);
 }
```

```
 get [TRACKED_OBJECT]() {
  return this.#tracked_elements;
 }

 get $length() {
  return get(this.#tracked_index);
 }

 set $length(length) {
  var block = scope();
  var tracked_elements = this.#tracked_elements;

  if (length !== this.$length) {
   for (var i = 0; i < tracked_elements.length; i++) {
    increment(tracked_elements[i], block);
   }
   this.length = length;
   tracked_elements.length = length;

   return true;
  }
  return false;
 }

 set length(_) {
  throw new Error('Cannot set length on RippleArray, use $length instead');
 }

 toJSON() {
  this.$length;
  return get_all_elements(this);
 }
}

export function get_all_elements(array) {
 var tracked_elements = array[TRACKED_OBJECT];
 var arr = [];

 for (var i = 0; i < tracked_elements.length; i++) {
  get(tracked_elements[i]);
  arr.push(array[i]);
 }

 return arr;
}

export function array(...elements) {
 return new RippleArray(...elements);
}
```

src/runtime/index.js

```
import { destroy_block, root } from './internal/client/blocks.js';
import { handle_root_events } from './internal/client/events.js';
import { init_operations } from './internal/client/operations.js';
import { active_block } from './internal/client/runtime.js';
import { create_anchor } from './internal/client/utils.js';
```

```
// Re-export JSX runtime functions for jsxImportSource: "ripple"
export { jsx, jsxs, Fragment } from '../jsx-runtime.js';

export function mount(component, options) {
 init_operations();

 const props = options.props || {};
 const target = options.target;
 const anchor = create_anchor();
 target.append(anchor);

 const cleanup_events = handle_root_events(target);

 const _root = root(() => {
  component(anchor, props, active_block);
 });

 return () => {
  cleanup_events();
  destroy_block(_root);
  target.removeChild(anchor_node);
 };
}

export { flush_sync as flushSync, untrack, deferred } from './internal/client/runtime.js';

export { array } from './array.js';

export { keyed } from './internal/client/for.js';

export { user_effect as effect } from './internal/client/blocks.js';

export { Portal } from './internal/client/portal.js';
```

src/utils/ast.js

```
import * as b from './builders.js';

export function object(expression) {
 while (expression.type === 'MemberExpression') {
  expression = /** @type {ESTree.MemberExpression | ESTree.Identifier} */ (expression.object);
 }

 if (expression.type !== 'Identifier') {
  return null;
 }

 return expression;
}

export function unwrap_pattern(pattern, nodes = []) {
 switch (pattern.type) {
  case 'Identifier':
   nodes.push(pattern);
   break;

  case 'MemberExpression':
   // member expressions can be part of an assignment pattern, but not a binding pattern
```

```
    // see https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment#binding_and_assignment
    nodes.push(pattern);
    break;

  case 'ObjectPattern':
    for (const prop of pattern.properties) {
      if (prop.type === 'RestElement') {
        unwrap_pattern(prop.argument, nodes);
      } else {
        unwrap_pattern(prop.value, nodes);
      }
    }

    break;

  case 'ArrayPattern':
    for (const element of pattern.elements) {
      if (element) unwrap_pattern(element, nodes);
    }

    break;

  case 'RestElement':
    unwrap_pattern(pattern.argument, nodes);
    break;

  case 'AssignmentPattern':
    unwrap_pattern(pattern.left, nodes);
    break;
 }

 return nodes;
}

export function extract_identifiers(pattern) {
 return unwrap_pattern(pattern, []).filter((node) => node.type === 'Identifier');
}

export function extract_paths(param) {
 return _extract_paths(
  [],
  param,
  (node) => /** @type {ESTree.Identifier | ESTree.MemberExpression} */ (node),
  (node) => /** @type {ESTree.Identifier | ESTree.MemberExpression} */ (node),
  false
 );
}

function _extract_paths(assignments = [], param, expression, update_expression, has_default_value) {
 switch (param.type) {
  case 'Identifier':
  case 'MemberExpression':
    assignments.push({
      node: param,
      is_rest: false,
      has_default_value,
      expression,
      update_expression
    });
```

```
    break;

  case 'ObjectPattern':
    for (const prop of param.properties) {
      if (prop.type === 'RestElement') {
        /** @type {DestructuredAssignment['expression']} */
        const rest_expression = (object) => {
          /** @type {ESTree.Expression[]} */
          const props = [];

          for (const p of param.properties) {
            if (p.type === 'Property' && p.key.type !== 'PrivateIdentifier') {
              if (p.key.type === 'Identifier' && !p.computed) {
                props.push(b.literal(p.key.name));
              } else if (p.key.type === 'Literal') {
                props.push(b.literal(String(p.key.value)));
              } else {
                props.push(b.call('String', p.key));
              }
            }
          }

          return b.call('$.exclude_from_object', expression(object), b.array(props));
        };

        if (prop.argument.type === 'Identifier') {
          assignments.push({
            node: prop.argument,
            is_rest: true,
            has_default_value,
            expression: rest_expression,
            update_expression: rest_expression
          });
        } else {
          _extract_paths(
            assignments,
            prop.argument,
            rest_expression,
            rest_expression,
            has_default_value
          );
        }
      } else {
        /** @type {DestructuredAssignment['expression']} */
        const object_expression = (object) =>
          b.member(expression(object), prop.key, prop.computed || prop.key.type !== 'Identifier');
        _extract_paths(
          assignments,
          prop.value,
          object_expression,
          object_expression,
          has_default_value
        );
      }
    }

    break;

  case 'ArrayPattern':
    for (let i = 0; i < param.elements.length; i += 1) {
```

```
     const element = param.elements[i];
     if (element) {
      if (element.type === 'RestElement') {
       /** @type {DestructuredAssignment['expression']} */
       const rest_expression = (object) =>
        b.call(b.member(expression(object), 'slice'), b.literal(i));
       if (element.argument.type === 'Identifier') {
        assignments.push({
         node: element.argument,
         is_rest: true,
         has_default_value,
         expression: rest_expression,
         update_expression: rest_expression
        });
       } else {
        _extract_paths(
         assignments,
         element.argument,
         rest_expression,
         rest_expression,
         has_default_value
        );
       }
      } else {
       /** @type {DestructuredAssignment['expression']} */
       const array_expression = (object) => b.member(expression(object), b.literal(i), true);
       _extract_paths(
        assignments,
        element,
        array_expression,
        array_expression,
        has_default_value
       );
      }
     }
    }

    break;

   case 'AssignmentPattern': {
    /** @type {DestructuredAssignment['expression']} */
    const fallback_expression = (object) => build_fallback(expression(object), param.right);

    if (param.left.type === 'Identifier') {
     assignments.push({
      node: param.left,
      is_rest: false,
      has_default_value: true,
      expression: fallback_expression,
      update_expression
     });
    } else {
     _extract_paths(assignments, param.left, fallback_expression, update_expression, true);
    }

    break;
   }
  }

  return assignments;
```

```
}

/**
 * @param {ESTree.AssignmentOperator} operator
 * @param {ESTree.Identifier | ESTree.MemberExpression} left
 * @param {ESTree.Expression} right
 */
export function build_assignment_value(operator, left, right) {
 return operator === '='
  ? right
  : // turn something like x += 1 into x = x + 1
   b.binary(/** @type {ESTree.BinaryOperator} */ (operator.slice(0, -1)), left, right);
}
```

src/utils/builders.js

```
/** @import * as ESTree from 'estree' */
import { regex_is_valid_identifier } from './patterns.js';
import { sanitize_template_string } from './sanitize_template_string.js';

/**
 * @param {Array<ESTree.Expression | ESTree.SpreadElement | null>} elements
 * @returns {ESTree.ArrayExpression}
 */
export function array(elements = []) {
 return { type: 'ArrayExpression', elements };
}

/**
 * @param {Array<ESTree.Pattern | null>} elements
 * @returns {ESTree.ArrayPattern}
 */
export function array_pattern(elements) {
 return { type: 'ArrayPattern', elements };
}

/**
 * @param {ESTree.Pattern} left
 * @param {ESTree.Expression} right
 * @returns {ESTree.AssignmentPattern}
 */
export function assignment_pattern(left, right) {
 return { type: 'AssignmentPattern', left, right };
}

/**
 * @param {Array<ESTree.Pattern>} params
 * @param {ESTree.BlockStatement | ESTree.Expression} body
 * @returns {ESTree.ArrowFunctionExpression}
 */
export function arrow(params, body, async = false) {
 return {
  type: 'ArrowFunctionExpression',
  params,
  body,
  expression: body.type !== 'BlockStatement',
  generator: false,
  async,
```

```
  metadata: /** @type {any} */ (null) // should not be used by codegen
 };
}

/**
 * @param {ESTree.AssignmentOperator} operator
 * @param {ESTree.Pattern} left
 * @param {ESTree.Expression} right
 * @returns {ESTree.AssignmentExpression}
 */
export function assignment(operator, left, right) {
 return { type: 'AssignmentExpression', operator, left, right };
}

/**
 * @template T
 * @param {T & ESTree.BaseFunction} func
 * @returns {T & ESTree.BaseFunction}
 */
export function async(func) {
 return { ...func, async: true };
}

/**
 * @param {ESTree.Expression} argument
 * @returns {ESTree.AwaitExpression}
 */
function await_builder(argument) {
 return { type: 'AwaitExpression', argument };
}

/**
 * @param {ESTree.BinaryOperator} operator
 * @param {ESTree.Expression} left
 * @param {ESTree.Expression} right
 * @returns {ESTree.BinaryExpression}
 */
export function binary(operator, left, right) {
 return { type: 'BinaryExpression', operator, left, right };
}

/**
 * @param {ESTree.Statement[]} body
 * @returns {ESTree.BlockStatement}
 */
export function block(body) {
 return { type: 'BlockStatement', body };
}

/**
 * @param {string} name
 * @param {ESTree.Statement} body
 * @returns {ESTree.LabeledStatement}
 */
export function labeled(name, body) {
 return { type: 'LabeledStatement', label: id(name), body };
}

/**
 * @param {string | ESTree.Expression} callee
```

```
 * @param {...(ESTree.Expression | ESTree.SpreadElement | false | undefined)} args
 * @returns {ESTree.CallExpression}
 */
export function call(callee, ...args) {
 if (typeof callee === 'string') callee = id(callee);
 args = args.slice();

 // replacing missing arguments with `undefined`, unless they're at the end in which case remove the
m
 let i = args.length;
 let popping = true;
 while (i--) {
  if (!args[i]) {
   if (popping) {
    args.pop();
   } else {
    args[i] = id('undefined');
   }
  } else {
   popping = false;
  }
 }

 return {
  type: 'CallExpression',
  callee,
  arguments: /** @type {Array<ESTree.Expression | ESTree.SpreadElement>} */ (args),
  optional: false
 };
}

/**
 * @param {string | ESTree.Expression} callee
 * @param {...ESTree.Expression} args
 * @returns {ESTree.ChainExpression}
 */
export function maybe_call(callee, ...args) {
 const expression = /** @type {ESTree.SimpleCallExpression} */ (call(callee, ...args));
 expression.optional = true;

 return {
  type: 'ChainExpression',
  expression
 };
}

/**
 * @param {ESTree.UnaryOperator} operator
 * @param {ESTree.Expression} argument
 * @returns {ESTree.UnaryExpression}
 */
export function unary(operator, argument) {
 return { type: 'UnaryExpression', argument, operator, prefix: true };
}

/**
 * @param {ESTree.Expression} test
 * @param {ESTree.Expression} consequent
 * @param {ESTree.Expression} alternate
 * @returns {ESTree.ConditionalExpression}
```

```
 */
export function conditional(test, consequent, alternate) {
 return { type: 'ConditionalExpression', test, consequent, alternate };
}

/**
 * @param {ESTree.LogicalOperator} operator
 * @param {ESTree.Expression} left
 * @param {ESTree.Expression} right
 * @returns {ESTree.LogicalExpression}
 */
export function logical(operator, left, right) {
 return { type: 'LogicalExpression', operator, left, right };
}

/**
 * @param {'const' | 'let' | 'var'} kind
 * @param {ESTree.VariableDeclarator[]} declarations
 * @returns {ESTree.VariableDeclaration}
 */
export function declaration(kind, declarations) {
 return {
  type: 'VariableDeclaration',
  kind,
  declarations
 };
}

/**
 * @param {ESTree.Pattern | string} pattern
 * @param {ESTree.Expression} [init]
 * @returns {ESTree.VariableDeclarator}
 */
export function declarator(pattern, init) {
 if (typeof pattern === 'string') pattern = id(pattern);
 return { type: 'VariableDeclarator', id: pattern, init };
}

/** @type {ESTree.EmptyStatement} */
export const empty = {
 type: 'EmptyStatement'
};

/**
 * @param {ESTree.Expression | ESTree.MaybeNamedClassDeclaration | ESTree.MaybeNamedFunctionDeclarat
ion} declaration
 * @returns {ESTree.ExportDefaultDeclaration}
 */
export function export_default(declaration) {
 return { type: 'ExportDefaultDeclaration', declaration };
}

/**
 * @param {ESTree.Identifier} id
 * @param {ESTree.Pattern[]} params
 * @param {ESTree.BlockStatement} body
 * @returns {ESTree.FunctionDeclaration}
 */
export function function_declaration(id, params, body) {
 return {
```

```
  type: 'FunctionDeclaration',
  id,
  params,
  body,
  generator: false,
  async: false,
  metadata: /** @type {any} */ (null) // should not be used by codegen
 };
}

/**
 * @param {string} name
 * @param {ESTree.Statement[]} body
 * @returns {ESTree.Property & { value: ESTree.FunctionExpression}}}
 */
export function get(name, body) {
 return prop('get', key(name), function_builder(null, [], block(body)));
}

/**
 * @param {string} name
 * @returns {ESTree.Identifier}
 */
export function id(name) {
 return { type: 'Identifier', name };
}

/**
 * @param {string} name
 * @returns {ESTree.PrivateIdentifier}
 */
export function private_id(name) {
 return { type: 'PrivateIdentifier', name };
}

/**
 * @param {string} local
 * @returns {ESTree.ImportNamespaceSpecifier}
 */
function import_namespace(local) {
 return {
  type: 'ImportNamespaceSpecifier',
  local: id(local)
 };
}

/**
 * @param {string} name
 * @param {ESTree.Expression} value
 * @returns {ESTree.Property}
 */
export function init(name, value) {
 return prop('init', key(name), value);
}

/**
 * @param {string | boolean | null | number | RegExp} value
 * @returns {ESTree.Literal}
 */
export function literal(value) {
```

```
 // @ts-expect-error we don't want to muck around with bigint here
 return { type: 'Literal', value };
}

/**
 * @param {ESTree.Expression | ESTree.Super} object
 * @param {string | ESTree.Expression | ESTree.PrivateIdentifier} property
 * @param {boolean} computed
 * @param {boolean} optional
 * @returns {ESTree.MemberExpression}
 */
export function member(object, property, computed = false, optional = false) {
 if (typeof property === 'string') {
  property = id(property);
 }

 return { type: 'MemberExpression', object, property, computed, optional };
}

/**
 * @param {string} path
 * @returns {ESTree.Identifier | ESTree.MemberExpression}
 */
export function member_id(path) {
 const parts = path.split('.');

 /** @type {ESTree.Identifier | ESTree.MemberExpression} */
 let expression = id(parts[0]);

 for (let i = 1; i < parts.length; i += 1) {
  expression = member(expression, id(parts[i]));
 }
 return expression;
}

/**
 * @param {Array<ESTree.Property | ESTree.SpreadElement>} properties
 * @returns {ESTree.ObjectExpression}
 */
export function object(properties) {
 return { type: 'ObjectExpression', properties };
}

/**
 * @param {Array<ESTree.RestElement | ESTree.AssignmentProperty | ESTree.Property>} properties
 * @returns {ESTree.ObjectPattern}
 */
export function object_pattern(properties) {
 // @ts-expect-error the types appear to be wrong
 return { type: 'ObjectPattern', properties };
}

/**
 * @template {ESTree.Expression} Value
 * @param {'init' | 'get' | 'set'} kind
 * @param {ESTree.Expression} key
 * @param {Value} value
 * @param {boolean} computed
 * @returns {ESTree.Property & { value: Value }}
 */
```

```
export function prop(kind, key, value, computed = false) {
 return { type: 'Property', kind, key, value, method: false, shorthand: false, computed };
}

/**
 * @param {ESTree.Expression | ESTree.PrivateIdentifier} key
 * @param {ESTree.Expression | null | undefined} value
 * @param {boolean} computed
 * @param {boolean} is_static
 * @returns {ESTree.PropertyDefinition}
 */
export function prop_def(key, value, computed = false, is_static = false) {
 return { type: 'PropertyDefinition', key, value, computed, static: is_static };
}

/**
 * @param {string} cooked
 * @param {boolean} tail
 * @returns {ESTree.TemplateElement}
 */
export function quasi(cooked, tail = false) {
 const raw = sanitize_template_string(cooked);
 return { type: 'TemplateElement', value: { raw, cooked }, tail };
}

/**
 * @param {ESTree.Pattern} argument
 * @returns {ESTree.RestElement}
 */
export function rest(argument) {
 return { type: 'RestElement', argument };
}

/**
 * @param {ESTree.Expression[]} expressions
 * @returns {ESTree.SequenceExpression}
 */
export function sequence(expressions) {
 return { type: 'SequenceExpression', expressions };
}

/**
 * @param {string} name
 * @param {ESTree.Statement[]} body
 * @returns {ESTree.Property & { value: ESTree.FunctionExpression}}
 */
export function set(name, body) {
 return prop('set', key(name), function_builder(null, [id('$$value')], block(body)));
}

/**
 * @param {ESTree.Expression} argument
 * @returns {ESTree.SpreadElement}
 */
export function spread(argument) {
 return { type: 'SpreadElement', argument };
}

/**
 * @param {ESTree.Expression} expression
```

```
 * @returns {ESTree.ExpressionStatement}
 */
export function stmt(expression) {
 return { type: 'ExpressionStatement', expression };
}

/**
 * @param {ESTree.TemplateElement[]} elements
 * @param {ESTree.Expression[]} expressions
 * @returns {ESTree.TemplateLiteral}
 */
export function template(elements, expressions) {
 return { type: 'TemplateLiteral', quasis: elements, expressions };
}

/**
 * @param {ESTree.Expression | ESTree.BlockStatement} expression
 * @param {boolean} [async]
 * @returns {ESTree.Expression}
 */
export function thunk(expression, async = false) {
 const fn = arrow([], expression);
 if (async) fn.async = true;
 return unthunk(fn);
}

/**
 * Replace "(arg) => func(arg)" to "func"
 * @param {ESTree.Expression} expression
 * @returns {ESTree.Expression}
 */
export function unthunk(expression) {
 if (
  expression.type === 'ArrowFunctionExpression' &&
  expression.async === false &&
  expression.body.type === 'CallExpression' &&
  expression.body.callee.type === 'Identifier' &&
  expression.params.length === expression.body.arguments.length &&
  expression.params.every((param, index) => {
   const arg = /** @type {ESTree.SimpleCallExpression} */ (expression.body).arguments[index];
   return param.type === 'Identifier' && arg.type === 'Identifier' && param.name === arg.name;
  })
 ) {
  return expression.body.callee;
 }
 return expression;
}

/**
 *
 * @param {string | ESTree.Expression} expression
 * @param  {...ESTree.Expression} args
 * @returns {ESTree.NewExpression}
 */
function new_builder(expression, ...args) {
 if (typeof expression === 'string') expression = id(expression);

 return {
  callee: expression,
  arguments: args,
```

```
  type: 'NewExpression'
 };
}

/**
 * @param {ESTree.UpdateOperator} operator
 * @param {ESTree.Expression} argument
 * @param {boolean} prefix
 * @returns {ESTree.UpdateExpression}
 */
export function update(operator, argument, prefix = false) {
 return { type: 'UpdateExpression', operator, argument, prefix };
}

/**
 * @param {ESTree.Expression} test
 * @param {ESTree.Statement} body
 * @returns {ESTree.DoWhileStatement}
 */
export function do_while(test, body) {
 return { type: 'DoWhileStatement', test, body };
}

const true_instance = literal(true);
const false_instance = literal(false);
const null_instane = literal(null);

/** @type {ESTree.DebuggerStatement} */
const debugger_builder = {
 type: 'DebuggerStatement'
};

/** @type {ESTree.ThisExpression} */
const this_instance = {
 type: 'ThisExpression'
};

/**
 * @param {string | ESTree.Pattern} pattern
 * @param { ESTree.Expression} [init]
 * @returns {ESTree.VariableDeclaration}
 */
function let_builder(pattern, init) {
 return declaration('let', [declarator(pattern, init)]);
}

/**
 * @param {string | ESTree.Pattern} pattern
 * @param { ESTree.Expression} init
 * @returns {ESTree.VariableDeclaration}
 */
function const_builder(pattern, init) {
 return declaration('const', [declarator(pattern, init)]);
}

/**
 * @param {string | ESTree.Pattern} pattern
 * @param { ESTree.Expression} [init]
 * @returns {ESTree.VariableDeclaration}
 */
```

```
function var_builder(pattern, init) {
 return declaration('var', [declarator(pattern, init)]);
}

/**
 *
 * @param {ESTree.VariableDeclaration | ESTree.Expression | null} init
 * @param {ESTree.Expression} test
 * @param {ESTree.Expression} update
 * @param {ESTree.Statement} body
 * @returns {ESTree.ForStatement}
 */
function for_builder(init, test, update, body) {
 return { type: 'ForStatement', init, test, update, body };
}

/**
 * @param {ESTree.VariableDeclaration | ESTree.Pattern} left
 * @param {ESTree.Expression} right
 * @param {ESTree.Statement} body
 * @param {boolean} [await_flag]
 * @returns {ESTree.ForOfStatement}
 */
export function for_of(left, right, body, await_flag = false) {
 return { type: 'ForOfStatement', left, right, body, await: await_flag };
}

/**
 *
 * @param {'constructor' | 'method' | 'get' | 'set'} kind
 * @param {ESTree.Expression | ESTree.PrivateIdentifier} key
 * @param {ESTree.Pattern[]} params
 * @param {ESTree.Statement[]} body
 * @param {boolean} computed
 * @param {boolean} is_static
 * @returns {ESTree.MethodDefinition}
 */
export function method(kind, key, params, body, computed = false, is_static = false) {
 return {
  type: 'MethodDefinition',
  key,
  kind,
  value: function_builder(null, params, block(body)),
  computed,
  static: is_static
 };
}

/**
 *
 * @param {ESTree.Identifier | null} id
 * @param {ESTree.Pattern[]} params
 * @param {ESTree.BlockStatement} body
 * @returns {ESTree.FunctionExpression}
 */
function function_builder(id, params, body) {
 return {
  type: 'FunctionExpression',
  id,
  params,
```

```
  body,
  generator: false,
  async: false,
  metadata: /** @type {any} */ (null) // should not be used by codegen
 };
}

/**
 * @param {ESTree.Expression} test
 * @param {ESTree.Statement} consequent
 * @param {ESTree.Statement} [alternate]
 * @returns {ESTree.IfStatement}
 */
function if_builder(test, consequent, alternate) {
 return { type: 'IfStatement', test, consequent, alternate };
}

/**
 * @param {string} as
 * @param {string} source
 * @returns {ESTree.ImportDeclaration}
 */
export function import_all(as, source) {
 return {
  type: 'ImportDeclaration',
  source: literal(source),
  specifiers: [import_namespace(as)]
 };
}

/**
 * @param {Array<[string, string]>} parts
 * @param {string} source
 * @returns {ESTree.ImportDeclaration}
 */
export function imports(parts, source) {
 return {
  type: 'ImportDeclaration',
  source: literal(source),
  specifiers: parts.map((p) => ({
   type: 'ImportSpecifier',
   imported: id(p[0]),
   local: id(p[1])
  }))
 };
}

/**
 * @param {ESTree.Expression | null} argument
 * @returns {ESTree.ReturnStatement}
 */
function return_builder(argument = null) {
 return { type: 'ReturnStatement', argument };
}

/**
 * @param {string} str
 * @returns {ESTree.ThrowStatement}
 */
export function throw_error(str) {
```

```
 return {
  type: 'ThrowStatement',
  argument: new_builder('Error', literal(str))
 };
}



/**
 * @param {string} name
 * @returns {ESTree.Expression}
 */
export function key(name) {
 return regex_is_valid_identifier.test(name) ? id(name) : literal(name);
}

/**
 * @param {ESTree.JSXIdentifier | ESTree.JSXNamespacedName} name
 * @param {ESTree.Literal | ESTree.JSXExpressionContainer | null} value
 * @returns {ESTree.JSXAttribute}
 */
export function jsx_attribute(name, value = null) {
 return {
  type: 'JSXAttribute',
  name,
  value
 };
}

/**
 * @param {ESTree.JSXIdentifier | ESTree.JSXMemberExpression | ESTree.JSXNamespacedName} name
 * @param {Array<ESTree.JSXAttribute | ESTree.JSXSpreadAttribute>} attributes
 * @param {Array<ESTree.JSXText | ESTree.JSXExpressionContainer | ESTree.JSXSpreadChild | ESTree.JSX
Element | ESTree.JSXFragment>} children
 * @param {boolean} self_closing
 * @returns {{ element: ESTree.JSXElement, opening_element: ESTree.JSXOpeningElement }}
 */
export function jsx_element(name, attributes = [], children = [], self_closing = false, closing_name
= name) {
 const opening_element = {
  type: 'JSXOpeningElement',
  name,
  attributes,
  selfClosing: self_closing
 };

 const element = {
  type: 'JSXElement',
  openingElement: opening_element,
  closingElement: self_closing ? null : {
   type: 'JSXClosingElement',
   name: closing_name
  },
  children
 };

 return element;
}

/**
 * @param {ESTree.Expression | ESTree.JSXEmptyExpression} expression
```

```
 * @returns {ESTree.JSXExpressionContainer}
 */
export function jsx_expression_container(expression) {
 return {
  type: 'JSXExpressionContainer',
  expression
 };
}

/**
 * @param {string} name
 * @returns {ESTree.JSXIdentifier}
 */
export function jsx_id(name) {
 return {
  type: 'JSXIdentifier',
  name
 };
}

export {
 await_builder as await,
 let_builder as let,
 const_builder as const,
 var_builder as var,
 true_instance as true,
 false_instance as false,
 for_builder as for,
 function_builder as function,
 return_builder as return,
 if_builder as if,
 this_instance as this,
 null_instane as null,
 debugger_builder as debugger
};
```

src/utils/patterns.js

```
export const regex_whitespace = /\s/;
export const regex_whitespaces = /\s+/;
export const regex_starts_with_newline = /^\r?\n/;
export const regex_starts_with_whitespace = /^\s/;
export const regex_starts_with_whitespaces = /^[ \t\r\n]+/;
export const regex_ends_with_whitespace = /\s$/;
export const regex_ends_with_whitespaces = /[ \t\r\n]+$/;
/** Not \S because that also removes explicit whitespace defined through things like ` ` */
export const regex_not_whitespace = /[^ \t\r\n]/;
/** Not \s+ because that also includes explicit whitespace defined through things like ` ` */
export const regex_whitespaces_strict = /[ \t\n\r\f]+/g;

export const regex_only_whitespaces = /^[ \t\n\r\f]+$/;

export const regex_not_newline_characters = /[^\n]/g;

export const regex_is_valid_identifier = /^[a-zA-Z_$][a-zA-Z_$0-9]*$/;
// used in replace all to remove all invalid chars from a literal identifier
export const regex_invalid_identifier_chars = /([^a-zA-Z_$]|[^a-zA-Z0-9_$])/g;
```

```
export const regex_starts_with_vowel = /^[aeiou]/;
export const regex_heading_tags = /^h[1-6]$/;
export const regex_illegal_attribute_character = /(^[0-9-.])|[\^$@%&#?!|()[\]{}^*+~;]/;
```

src/utils/sanitize_template_string.js

```
/**
 * @param {string} str
 * @returns {string}
 */
export function sanitize_template_string(str) {
 return str.replace(/(`|\${|\\)/g, '\\$1');
}
```

src/ai.js

```
// import { anthropic } from '@ai-sdk/anthropic';
// import { generateText } from 'ai';

// const default_prompt = `
//   Ripple is a web-based JavaScript framework for building user interfaces. It's syntax and design
is inspired by React and Svelte 5.
//   It uses JSX for templating inside '.ripple' modules. These modules allow for custom syntax that
is not JavaScript compliant.

//   One of the core differences is that it allows for a new type of JavaScript declaration which is
a 'component', which is like a 'function' but is only allowed in '.ripple' modules:

//   \`\`\`js
//     component HelloComponent(props) {
//         const title = 'Hello ';

//         <div>{title + props.name}</div>;
//     }
//   \`\`\`

//   Another difference is that 'component' declaration bodies allow for JSX templating. Except this
JSX templating isn't "expression" based, but rather "statement" based. That
//   means that 'return' is not valid syntax in a component declaration body. Nor is creating a vari
able that references JSX. Instead, JSX is directly written in the body of the component declaration.
//   This means that the ordering is important, as JSX that is written first will be rendered first.
This is different from React.

//   Another difference from JSX in other frameworks is that JSXText is not allowed here. That's bec
ause JSX is now statement based, and not expression based. This means that all JSX must be wrapped i
n a JSXExpressionContainer.

//   For example, this is invalid Ripple code:

//   \`\`\`js
//     <button>=</button>
//   \`\`\`

//   The correct version is:
```

```
//   \`\`\`js
//     <button>{"="}</button>
//   \`\`\`

//   Another core difference is that Ripple defines reactive variables by their usage of a "$" prefi
x. If the variable declaration does not have a dollar prefix, it is not reactive.

//   \`\`\`js
//     component HelloComponent(props) {
//         let $count = 0;

//         <div>{$count}</div>;
//         <button onClick={() => $count++}>{"Increment"}</button>;
//     }
//   \`\`\`

//   Object properties can also be reactive if the property name starts with a "$" prefix.

//   \`\`\`js
//     component HelloComponent(props) {
//         let state = { $count: 0 };

//         <div>{state.$count}</div>;
//         <button onClick={() => state.$count++}>{"Increment"}</button>;
//     }
//   \`\`\`

//   Ripple doesn't allow for inline expressions with JSX for conditionals or for collections such a
s arrays or objects.
//   Instead, prefer using normal JavaScript logic where you have a "if" or "for" statement that wra
ps the JSX.

//   Here is valid Ripple code:

//   \`\`\`js
//     export component Counter() {
//         let $count = 0;

//         if ($count > 5) {
//           <div>{$count}</div>;
//         }

//         <div>
//           if ($count > 5) {
//             <div>{$count}</div>;
//           }
//         </div>;

//         for (const item of items) {
//           <div>{item}</div>;
//         }

//         <ul>
//           for (const item of items) {
//             <li>{item}</li>;
//           }
//         </ul>;
//     }
//   \`\`\`
```

```
//   Ripple allows for shorthand props on components, so '<Child state={state} />' can be written as
'<Child {state} />'.

//   Ripple also allows for a singular "<style>" JSX element at the top level of the component decla
ration body. This is used for styling any JSX elements within the component.
//   The style element can contain any valid CSS, and can also contain CSS variables. CSS variables
are defined with a "--" prefix. This is the preferred way of doing styling over inline styles.

//   If inline styles are to be used, then they should be done using the HTML style attribute approa
ch rather than the JSX style attribute property approach.

//   In Ripple variables that are created with an identifier that starts with a "$" prefix are consi
dered reactive. If declaration init expression also references reactive variables, or function expre
ssions, then
//   this type of variable is considered "computed". Computed reactive declarations will re-run when
any of the reactive variables they reference change. If this is not desired then the "untrack" funct
ion call should
//   be used to prevent reactivity.

//   \`\`\`js
//     import { untrack } from 'ripple';

//     component Counter({ $initial }) {
//         let $count = untrack(() => $initial);
//     }
//   \`\`\`

//   An important part of Ripple's reactivity model is that passing reactivity between boundaries ca
n only happen via two ways:
//   - the usage of closures, where a value is referenced in a function or property getter
//   - the usage of objects and/or arrays, where the object or array is passed as a property with a
"$" prefix so its reactivity is kept

//   For example if you were to create a typical Ripple hook function, then you should pass any reac
tive values through using objects. Otherwise, the
//   hook will act as a computed function and re-run every time the reactive value changes — which i
s likely not the desired behaviour of a "hook" function.

//   \`\`\`js
//     function useCounter(initial) {
//       let $count = initial;
//       const $double = $count * 2;

//       const increment = () => $count++;

//       return { $double, increment };
//     }

//     component Counter({ $count }) {
//       const { $double, increment } = useCounter($count);

//       <button onClick={increment}>{"Increment"}</button>;
//       <div>{$double}</div>;
//     }
//   \`\`\`

//   If a value needs to be mutated from within a hook, then it should be referenced by the hook in
its object form instead:

//   \`\`\`js
```

```
//     function useCounter(state) {
//       const $double = state.$count * 2;

//       const increment = () => state.$count++;

//       return { $double, increment };
//     }

//     component Counter({ $count }) {
//       let $count = 0;

//       const { $double, increment } = useCounter({ $count });

//       <button onClick={increment}>{"Increment"}</button>;
//       <div>{$double}</div>;
//     }
//   \`\`\`

//   It should be noted that in this example, the "$count" inside the "Counter" component will not b
e mutated by the "increment" function.

//   If this is desired, then the call to "useCounter" needs to provide a getter and setter for the
"$count" value:

//   \`\`\`js
//     function useCounter(state) {
//       const $double = state.$count * 2;

//       const increment = () => state.$count++;

//       return { $double, increment };
//     }

//     component Counter({ $count }) {
//       let $count = 0;

//       const { $double, increment } = useCounter({ get $count() { return $count }, set $count(valu
e) { $count = value } });

//       <button onClick={increment}>{"Increment"}</button>;
//       <div>{$double}</div>;
//     }
//   \`\`\`

//   Normally, you shouldn't provide getters/setters in the object returned from a hook, especially
if the usage site intends to destruct the object.

//   Ripple also provides a way of handling Suspense and asynchronous data fetching. This requires t
wo parts:
//   - a "try" block, that has an "async" block that shows the fallback pending UI. These blocks can
only be used inside Ripple components
//   - an "await" that must happen at the top-level of the component body

//   Here is an example:

//   \`\`\`js
//     export component App() {
//       try {
//         <Child />;
//       } async {
```

```
//         <div>{"Loading..."}</div>;
//       }
//     }

//     component Child() {
//       const $pokemons = await fetch('https://pokeapi.co/api/v2/pokemon/').then((res) => res.json
());

//       for (const pokemon of $pokemons.results) {
//         <div>{pokemon.name}</div>;
//       }
//     }
//   \`\`\`

//   It's important that the transformed code never uses an async fetch() call inside an effect func
tion. This is an anti-pattern, instead the "await" expression should be used
//   directly inside the fragment or component body. Also when using "await" then loading states sho
uld be handled using the "try" and "async" blocks, so this isn't required in the
//   output code.

//   Ripple also supports "fragment" syntax, which is similar to the "component" syntax but allows f
or multiple arguments:

//   \`\`\`js
//     fragment foo() {
//       <div>{"Hello World"}</div>;
//     }

//     component App() {
//       {fragment foo()};
//     }
//   \`\`\`

//   Fragments can be seen as reactive functions that can take arguments and using the "{@fragment f
ragment(...args)}" syntax, they can be rendered as if they were JSX elements.

//   Ripple denotes attributes and properties on JSX elements as being reactive when they also have
a "$" prefix. This means that if a property is reactive, then the element will re-render when the pr
operty changes.

//   Ripple does not support both a non-reactive and reactive version of a prop — so having "$ref" a
nd "ref" is not allowed. If a prop could be possibly reactive, then it should always have a "$" pref
ix to ensure maximum compatibility.

//   There are also some special attributes that such as "$ref" and "$children" that always start wi
th a "$" prefix.

//   When creating an implicit children fragment from a JSX component, such as:

//   \`\`\`js
//     <ChildComponent>
//       {"Hello World"}
//     </ChildComponent>
//   \`\`\`

//   This can also be written as:

//   \`\`\`js
//     fragment $children() {
//       {"Hello World"};
```

```
//     }

//     <ChildComponent {$children} />;
//   \`\`\`

//   Which is the same as the previous example.

//   The "Hello world" will be passed as a "$children" prop to the "ChildComponent" and it will be o
f the type of "Fragment". Which means that it's not a string, or JSX element, but rather a special k
ind of thing.

//   To render a type of "Fragment" the {@fragment thing()} syntax should be used. This will render
the "thing" as if it was a JSX element. Here's an example:

//   \`\`\`js
//     component Child({ $children }) {
//       <div>
//         {@fragment $children()};
//       </div>;
//     }
//   \`\`\`

//   Ripple uses for...of blocks for templating over collections or lists. While loops, standard for
loops and while loops are not permitted in Ripple components or fragments.

//   For example, to render a list of items:

//   \`\`\`js
//     <ul>
//       for (const num of [10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]) {
//         <li>{num}</li>;
//       }
//     </ul>;
//   \`\`\`


// `;

// export async function validate_with_ai(source) {
//  const { text } = await generateText({
//      model: anthropic('claude-3-7-sonnet-20250219'),
//      messages: [
//          {
//              role: 'user',
//              content: default_prompt,
//              providerOptions: {
//                  anthropic: { cacheControl: { type: 'ephemeral' } }
//              }
//          },
//          {
//              role: 'user',
//              content: `Please validate the following Ripple code and provide feedback on any issu
es:\n\n${source}`
//          }
//      ]
//  });
//  return text;
// }
```

src/constants.js

```
export const TEMPLATE_FRAGMENT = 1;
export const TEMPLATE_USE_IMPORT_NODE = 1 << 1;
export const IS_CONTROLLED = 1 << 2;
```

src/jsx-runtime.d.ts

```
/**
 * Ripple JSX Runtime Type Definitions
 * Ripple components are imperative and don't return JSX elements
 */

// Ripple components don't return JSX elements - they're imperative
export type ComponentType<P = {}> = (props: P) => void;

/**
 * Create a JSX element (for elements with children)
 * In Ripple, this doesn't return anything - components are imperative
 */
export function jsx(
  type: string | ComponentType<any>,
  props?: any,
  key?: string | number | null
): void;

/**
 * Create a JSX element with static children (optimization for multiple children)
 * In Ripple, this doesn't return anything - components are imperative
 */
export function jsxs(
  type: string | ComponentType<any>,
  props?: any,
  key?: string | number | null
): void;

/**
 * JSX Fragment component
 * In Ripple, fragments are imperative and don't return anything
 */
export function Fragment(props: { $children?: any }): void;

// Base HTML attributes
interface HTMLAttributes {
  class?: string;
  className?: string;
  id?: string;
  style?: string | Record<string, string | number>;
  onClick?: (event: MouseEvent) => void;
  onInput?: (event: InputEvent) => void;
  onChange?: (event: Event) => void;
  $children?: any;
  [key: string]: any;
}

// Global JSX namespace for TypeScript
```

```
declare global {
  namespace JSX {
    // In Ripple, JSX expressions don't return elements - they're imperative
    type Element = void;

    interface IntrinsicElements {
      // HTML elements with basic attributes
      div: HTMLAttributes;
      span: HTMLAttributes;
      p: HTMLAttributes;
      h1: HTMLAttributes;
      h2: HTMLAttributes;
      h3: HTMLAttributes;
      h4: HTMLAttributes;
      h5: HTMLAttributes;
      h6: HTMLAttributes;
      button: HTMLAttributes & {
        type?: 'button' | 'submit' | 'reset';
        disabled?: boolean;
      };
      input: HTMLAttributes & {
        type?: string;
        value?: string | number;
        placeholder?: string;
        disabled?: boolean;
      };
      form: HTMLAttributes;
      a: HTMLAttributes & {
        href?: string;
        target?: string;
      };
      img: HTMLAttributes & {
        src?: string;
        alt?: string;
        width?: string | number;
        height?: string | number;
      };
      // Add more as needed...
      [elemName: string]: HTMLAttributes;
    }

    interface ElementChildrenAttribute {
      $children: {};
    }
  }
}
```

src/jsx-runtime.js

```
/**
 * Ripple JSX Runtime
 * This module provides the JSX runtime functions that TypeScript will automatically import
 * when using jsxImportSource: "ripple/jsx-runtime"
 */

/**
 * Create a JSX element (for elements with children)
 * In Ripple, components don't return values - they imperatively render to the DOM
```

```
 * @param {string | Function} type - Element type (tag name or component function)
 * @param {object} props - Element properties
 * @param {string} key - Element key (optional)
 * @returns {void} Ripple components don't return anything
 */
export function jsx(type, props, key) {
 // Ripple components are imperative - they don't return JSX elements
 // This is a placeholder for the actual Ripple rendering logic
 if (typeof type === 'function') {
  // Call the Ripple component function
  type(props);
 } else {
  // Handle DOM elements
  console.warn('DOM element rendering not implemented in jsx runtime:', type, props);
 }
}

/**
 * Create a JSX element with static children (optimization for multiple children)
 * @param {string | Function} type - Element type (tag name or component function)
 * @param {object} props - Element properties
 * @param {string} key - Element key (optional)
 * @returns {void} Ripple components don't return anything
 */
export function jsxs(type, props, key) {
 return jsx(type, props, key);
}

/**
 * JSX Fragment component
 * @param {object} props - Fragment props (should contain children)
 * @returns {void} Ripple fragments don't return anything
 */
export function Fragment(props) {
 // Ripple fragments are imperative
 console.warn('Fragment rendering not implemented in jsx runtime:', props);
}
```

types/index.d.ts

```
export type Fragment<T extends any[] = []> = (...args: T) => void;
```

package.json

```
{
 "name": "ripple",
 "description": "Ripple is a TypeScript UI framework for the web",
 "license": "MIT",
 "version": "0.0.1",
 "type": "module",
 "module": "src/runtime/index.js",
 "main": "src/runtime/index.js",
 "exports": {
  ".": {
   "types": "./types/index.d.ts",
```

```
    "worker": "./src/runtime/index.js",
    "browser": "./src/runtime/index.js",
    "default": "./src/runtime/index.js"
   },
   "./package.json": "./package.json",
   "./compiler": {
    "types": "./types/index.d.ts",
    "require": "./compiler/index.js",
    "default": "./src/compiler/index.js"
   },
   "./validator": {
    "types": "./types/index.d.ts",
    "require": "./validator/index.js",
    "default": "./src/validator/index.js"
   },
   "./internal/client": {
    "default": "./src/runtime/internal/client/index.js"
   },
   "./jsx-runtime": {
    "types": "./src/jsx-runtime.d.ts",
    "import": "./src/jsx-runtime.js",
    "default": "./src/jsx-runtime.js"
   }
  },
  "dependencies": {
   "@ai-sdk/anthropic": "^2.0.5",
   "@jridgewell/sourcemap-codec": "^1.5.5",
   "@types/estree": "^1.0.8",
   "acorn": "^8.15.0",
   "acorn-typescript": "^1.4.13",
   "esrap": "^2.1.0",
   "is-reference": "^3.0.3",
   "magic-string": "^0.30.17",
   "muggle-string": "^0.4.1",
   "zimmerframe": "^1.1.2"
  }
}
```

test-mappings.js