Struktur

```
root
├── compiler
│   ├── architecture.txt
│   ├── arch.ts
│   ├── assembler.ts
│   ├── ast.ts
│   ├── ast_visitor.ts
│   ├── checker.ts
│   ├── compiler.ts
│   ├── desugar
│   │   ├── compound_assign.ts
│   │   └── special_loop.ts
│   ├── info.ts
│   ├── lexer.ts
│   ├── lib.ts
│   ├── lowerer_locals.ts
│   ├── lowerer.ts
│   ├── main.ts
│   ├── mod.ts
│   ├── parser.ts
│   ├── resolver_syms.ts
│   ├── resolver.ts
│   ├── token.ts
│   └── vtype.ts
├── runtime
│   ├── actions.cpp
│   ├── actions.hpp
│   ├── alloc.cpp
│   ├── alloc.hpp
│   ├── arch.hpp
│   ├── compile_flags.txt
│   ├── instruction_size.cpp
│   ├── json.cpp
│   ├── json.hpp
│   ├── main.cpp
│   ├── rpc_server.cpp
│   ├── rpc_server.hpp
│   ├── to_json.cpp
│   ├── to_string.cpp
│   ├── value.cpp
│   ├── value.hpp
│   ├── vm.cpp
│   ├── vm.hpp
│   ├── vm_provider.cpp
│   └── vm_provider.hpp
├── slige-build-run-all.sh
├── slige-run.sh
├── stdlib.slg
└── web
    ├── main.ts
```

```
    ├── public
    │   ├── bundle.ts
    │   ├── index.html
    │   ├── src
    │   │   ├── code_coverage.ts
    │   │   ├── data.ts
    │   │   ├── flamegraph.ts
    │   │   └── index.ts
    │   └── style.css
    └── runtime.ts
```

slige-run.sh

```bash
#!/bin/bash

set -e

echo Text:
cat $1

echo Compiling $1...

deno run --allow-read --allow-write compiler/main.ts $1

echo Running out.slgbc...

./runtime/build/sliger run out.slgbc ${@:2}
```

web/public/style.css

```css
:root {
    color-scheme: light dark;
    font-family: sans;

    --bg-1: #2b2d31;
    --bg-2: #313338;
    --fg-2: #666666;

    --black: #211f1c;
    --black-transparent: #211f1caa;
    --white: #ecebe9;
    --white-transparent: #ecebe9aa;
    --code-status: var(--white);

    --code-bg: #282828;
    --code-fg: #fbf1c7;
    --code-linenr: #7c6f64;
}

* {
    box-sizing: border-box;
}
```

```css
body {
    margin: 0;
    height: 100vh;
    background-color: var(--black);
    color: var(--white);
}

body.status-waiting {
    --code-status: #e3b23c;
}

body.status-done {
    --code-status: #63a46c;
}

main {
    position: relative;
    flex: 1;
    padding: 1rem;
}
main > :not(#cover) {
    margin: 0 auto;
}

main #cover {
    position: absolute;
    inset: 0;
    display: flex;
    justify-content: center;
    align-items: center;
    background-color: var(--black-transparent);
    font-size: 2.5em;

    border-radius: 0.25rem;
    border: 2px solid var(--code-status);
}

.status-header {
    font-size: 1.75rem;
    padding: 1rem;
    background-color: var(--code-status);
    color: var(--black);
}
.status-header-content {
    margin: 0 auto;
    max-width: 1500px;
}

#views-nav {
    display: flex;
    flex-direction: column;
    padding: 1rem 0;
```

```
    border-radius: 0.25rem;
    border: 2px solid var(--code-status);
    gap: 0.5rem;
    min-width: 200px;
}

#views-nav input {
    display: none;
}

#views-nav label {
    display: inline-block;
    padding: 0.4em;
    padding-bottom: 0.2em;
    cursor: pointer;
    width: 100%;
    text-align: center;
}

#views-nav label:hover {
    background-color: rgba(255, 255, 255, 0.2);
}

#view {
    overflow: hidden;
}

#view .code-container {
    max-height: 100%;
    overflow: scroll;
    background-color: var(--code-bg);
    padding: 0.5rem;
    border-radius: 0.5rem;
}

#view .code-container pre {
    font-family: "Roboto Mono", monospace;
    font-weight: 500;
    color: var(--code-fg);
}

#view .code-container pre.code-lines {
    color: var(--code-linenr);
}

#view .code-container.code-coverage {
    max-height: calc(100% - 103px);
}

#view .code-container-inner {
    display: flex;
    font-size: 1rem;
    overflow: scroll;
```

```css
    max-height: 100%;
}

#view .code-lines {
    border-right: 1px solid currentcolor;
    padding-right: 0.5rem;
    margin: 0;
}

#view .code-source {
    width: 100%;
    padding-left: 0.5rem;
    margin: 0;
}

#view-nav div {
    width: 100%;
}

#views-nav input:checked + label {
    background-color: var(--code-status);
    color: var(--black);
    font-weight: bold;
}

#views-layout {
    display: flex;
    margin: 0 auto;
    padding: 1rem;
    gap: 1rem;
    max-width: 1500px;
    height: calc(100vh - 100px);
}

#covers-tooltip {
    z-index: 2;
    position: fixed;
    top: 0;
    left: 0;
    padding: 3px;
    border-radius: 3px;
    background-color: var(--black);
    border: 2px solid var(--code-status);
    border-radius: 0.25rem;
    color: #eee;
}

.coverage-radio {
    display: flex;
    flex-direction: row;
    justify-content: center;
}
```

```css
.coverage-radio-group {
    display: flex;
    justify-content: center;
    flex: 1;
    padding: 2rem 0.5rem;
    max-width: max-content;
}

.coverage-radio-group label {
    padding: 0.5rem;
    border-radius: 0.25rem;
    cursor: pointer;
    border: 2px solid var(--code-status);
    width: 280px;
    text-align: center;
}

.coverage-radio-group input:checked ~ label {
    background-color: var(--code-status);
    color: var(--black);
    font-weight: bold;
}

#flame-graph {
    width: min-content;
}

#flame-graph #fg-background {
    background-color: rgba(255, 255, 255, 0.1);
    padding: 0.5rem;
    border-radius: 0.5rem;
    width: min-content;
}

#flame-graph #canvas-div {
    width: 1004px;
    height: 504px;
    /*border: 2px solid rgb(240, 220, 200);*/
    padding: 4px;
}
#flame-graph canvas {
    z-index: 1;
    width: 1000px;
    height: 500px;
    position: absolute;
    image-rendering: pixelated;
    transform: translate(-2px, -2px);
}
#flame-graph #toolbar {
    margin: 20px;
    display: flex;
    flex-direction: row;
    justify-content: center;
```

```
    align-items: center;
}
#flame-graph #toolbar button {
    padding: 5px 20px;
    min-width: 100px;
}
#flame-graph #flame-graph-tooltip {
    z-index: 2;
    position: fixed;
    top: 0;
    left: 0;
    padding: 3px;
    border-radius: 3px;
    background-color: var(--bg-2);
    box-shadow: 2px 2px 2px black;
    color: #eee;
}
```

web/public/bundle.ts

```
import * as esbuild from "npm:esbuild";
import { denoPlugins } from "jsr:@luca/esbuild-deno-loader";

await esbuild.build({
    plugins: [...denoPlugins()],
    entryPoints: ["./src/index.ts"],
    outfile: "./dist/bundle.js",
    bundle: true,
    format: "esm",
});

esbuild.stop();
```

web/public/index.html

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">

        <link rel="preconnect" href="https://fonts.googleapis.com">
        <link rel="preconnect" href="https://fonts.gstatic.com"
crossorigin>
        <link href="https://fonts.googleapis.com/css2?
family=Roboto+Mono:ital,wght@0,100..700;1,100..700&display=swap"
rel="stylesheet">

        <link rel="stylesheet" href="style.css">
        <script src="dist/bundle.js" type="module" defer></script>
    </head>
    <body class="status-waiting">
        <div id="main-layout">
            <header class="status-header">
```

```
                <div class="status-header-content"><span>Status:</
span> <b id="status">Running</b>
            </header>
            <div id="views-layout">
                <nav id="views-nav">
                    <div>
                        <input type="radio" name="views"
value="source-code" id="source-code-radio" checked>
                        <label for="source-code-radio">Source
code</label>
                    </div>
                    <div>
                        <input type="radio" name="views"
value="code-coverage" id="code-coverage-radio">
                        <label for="code-coverage-radio">Code
coverage</label>
                    </div>
                    <div>
                        <input type="radio" name="views"
value="flame-graph" id="flame-graph-radio">
                        <label for="flame-graph-radio">Flame
graph</label>
                    </div>
                </nav>
                <main id="view">
                    <div id="flame-graph"></div>
                    <pre id="code-coverage"></pre>
                </main>
            </div>
        </div>
    </body>
</html>
```

web/public/src/data.ts

```
export type Status = {
    running: boolean;
};

export async function status(): Promise<Status> {
    return await fetch("/api/status")
        .then((v) => v.json())
        .then((v) => v.status);
}

export type FlameGraphNode = {
    fn: number;
    acc: number;
    parent: number;
    children: FlameGraphNode[];
};

export async function flameGraphData(): Promise<FlameGraphNode> {
```

```
    return await fetch("/api/flame-graph")
        .then((v) => v.json())
        .then((v) => v.flameGraph);
}

export type FlameGraphFnNames = { [key: number]: string };

export async function flameGraphFnNames():
Promise<FlameGraphFnNames> {
    return await fetch("/api/flame-graph-fn-names")
        .then((v) => v.json())
        .then((v) => v.fnNames);
}

export async function codeData(): Promise<string> {
    return await fetch("/api/source")
        .then((v) => v.json())
        .then((v) => v.text);
}

export type CodeCovEntry = {
    index: number;
    line: number;
    col: number;
    covers: number;
};

export async function codeCoverageData(): Promise<CodeCovEntry[]> {
    return await fetch("/api/code-coverage")
        .then((v) => v.json())
        .then((v) => v.codeCoverage);
}
```

web/public/src/flamegraph.ts

```
import * as data from "./data.ts";

export function loadFlameGraph(
    flameGraphData: data.FlameGraphNode,
    fnNames: data.FlameGraphFnNames,
    flameGraphDiv: HTMLDivElement,
) {
    flameGraphDiv.innerHTML = `
        <div id="fg-background">
            <div id="canvas-div">
                <canvas id="flame-graph-canvas"></canvas>
                <span id="flame-graph-tooltip" hidden></span>
            </div>
        </div>
        <div id="toolbar">
            <button id="flame-graph-reset">Reset</button>
        </div>
    `;
```

```
    const canvas = document.querySelector<HTMLCanvasElement>(
        "#flame-graph-canvas",
    )!;
    const resetButton = document.querySelector<HTMLButtonElement>(
        "#flame-graph-reset",
    )!;

    canvas.width = 1000;
    canvas.height = 500;

    const fnNameFont = "600 14px 'Roboto Mono'";

    const ctx = canvas.getContext("2d")!;
    ctx.font = fnNameFont;

    type CalcNode = {
        x: number;
        y: number;
        w: number;
        h: number;
        title: string;
        percent: string;
        fgNode: data.FlameGraphNode;
    };

    function calculateNodeRects(
        nodes: CalcNode[],
        node: data.FlameGraphNode,
        depth: number,
        totalAcc: data.FlameGraphNode["acc"],
        offsetAcc: data.FlameGraphNode["acc"],
    ) {
        const x = (offsetAcc / totalAcc) * canvas.width;
        const y = canvas.height - 30 * depth - 30;
        const w = ((node.acc + 1) / totalAcc) * canvas.width;
        const h = 30;

        const title = node.fn == 0
            ? "<program>"
            : fnNames[node.fn] ?? "<unknown>";
        const percent = `${(node.acc / totalAcc *
100).toFixed(1)}%`;
        nodes.push({ x, y, w, h, title, percent, fgNode: node });

        const totalChildrenAcc = node.children.reduce(
            (acc, child) => acc + child.acc,
            0,
        );
        let newOffsetAcc = offsetAcc + (node.acc -
totalChildrenAcc) / 2;
        for (const child of node.children) {
            calculateNodeRects(nodes, child, depth + 1, totalAcc,
```

```
newOffsetAcc);
            newOffsetAcc += child.acc;
        }
    }

    function drawTextCanvas(node: CalcNode): HTMLCanvasElement {
        const { w, h, title } = node;
        const textCanvas = document.createElement("canvas");
        textCanvas.width = Math.max(w - 8, 1);
        textCanvas.height = h;
        const textCtx = textCanvas.getContext("2d")!;
        textCtx.font = fnNameFont;
        textCtx.fillStyle = "black";
        textCtx.fillText(
            title,
            ((w - 10) / 2 - ctx.measureText(title).width / 2) + 5 -
4,
            20,
        );
        return textCanvas;
    }

    function renderNodes(nodes: CalcNode[], hoverFnId?: number) {
        for (const node of nodes) {
            const { x, y, w, h } = node;
            ctx.fillStyle = "rgb(227, 178, 60)";
            ctx.fillRect(x + 2, y + 2, w - 4, h - 4);

            const textCanvas = drawTextCanvas(node);
            ctx.drawImage(textCanvas, x + 4, y);

            const edgePadding = 4;
            const edgeWidth = 8;

            const leftGradient = ctx.createLinearGradient(
                x + 2 + edgePadding,
                0,
                x + 2 + edgeWidth,
                0,
            );
            leftGradient.addColorStop(1, "rgba(227, 178, 60,
0.0)");
            leftGradient.addColorStop(0, "rgba(227, 178, 60,
1.0)");
            ctx.fillStyle = leftGradient;
            ctx.fillRect(x + 2, y + 2, Math.min(edgeWidth, (w - 4)
/ 2), h - 4);

            const rightGradient = ctx.createLinearGradient(
                x + w - 2 - edgeWidth,
                0,
                x + w - 2 - edgePadding,
                0,
```

```
            );
            rightGradient.addColorStop(0, "rgba(227, 178, 60,
0.0)");
            rightGradient.addColorStop(1, "rgba(227, 178, 60,
1.0)");
            ctx.fillStyle = rightGradient;
            ctx.fillRect(
                x + w - 2 - Math.min(edgeWidth, (w - 4) / 2),
                y + 2,
                Math.min(edgeWidth, (w - 4) / 2),
                h - 4,
            );

            if (hoverFnId === node.fgNode.fn) {
                ctx.strokeStyle = canvas.style.backgroundColor;
                ctx.lineWidth = 1;
                ctx.strokeRect(x + 2, y + 2, w - 4, h - 4);
            }
        }
        const tooltip = document.getElementById("flame-graph-
tooltip")!;

        const mousemoveEvent = (e: MouseEvent) => {
            const x = e.offsetX;
            const y = e.offsetY;
            const node = nodes.find((node) =>
                x >= node.x && x < node.x + node.w && y >= node.y
&&
                y < node.y + node.h
            );

            if (!node) {
                tooltip.hidden = true;
                canvas.style.cursor = "default";
                return;
            }
            tooltip.innerText = `${node.title} ${node.percent}`;
            tooltip.style.left = `${e.clientX + 20}px`;
            tooltip.style.top = `${e.clientY + 20}px`;
            tooltip.hidden = false;
            canvas.style.cursor = "pointer";

            canvas.removeEventListener("mousemove",
mousemoveEvent);
            canvas.removeEventListener("mouseleave",
mouseleaveEvent);
            canvas.removeEventListener("mousedown",
mousedownEvent);
            ctx.clearRect(0, 0, canvas.width, canvas.height);
            renderNodes(nodes, node.fgNode.fn);
        };
        const mouseleaveEvent = () => {
            tooltip.hidden = true;
```

```
            canvas.style.cursor = "default";
        };
        const mousedownEvent = (e: MouseEvent) => {
            const x = e.offsetX;
            const y = e.offsetY;
            const node = nodes.find((node) =>
                x >= node.x && x < node.x + node.w && y >= node.y
&&
                y < node.y + node.h
            );
            if (!node) {
                return;
            }
            tooltip.hidden = true;
            canvas.style.cursor = "default";
            const newNodes: CalcNode[] = [];
            calculateNodeRects(
                newNodes,
                node.fgNode,
                0,
                node.fgNode.acc,
                0,
            );
            canvas.removeEventListener("mousemove",
mousemoveEvent);
            canvas.removeEventListener("mouseleave",
mouseleaveEvent);
            canvas.removeEventListener("mousedown",
mousedownEvent);
            ctx.clearRect(0, 0, canvas.width, canvas.height);
            renderNodes(newNodes);
        };

        canvas.addEventListener("mousemove", mousemoveEvent);
        canvas.addEventListener("mouseleave", mouseleaveEvent);
        canvas.addEventListener("mousedown", mousedownEvent);
    }

    resetButton.addEventListener("click", () => {
        const nodes: CalcNode[] = [];
        calculateNodeRects(nodes, flameGraphData, 0,
flameGraphData.acc, 0);
        renderNodes(nodes);
    });

    const nodes: CalcNode[] = [];
    calculateNodeRects(nodes, flameGraphData, 0,
flameGraphData.acc, 0);
    renderNodes(nodes);
}
```

web/public/src/index.ts

```
import { CodeCovRender, loadCodeCoverage } from "./
code_coverage.ts";
import * as data from "./data.ts";
import { loadFlameGraph } from "./flamegraph.ts";

function countLines(code: string) {
    let lines = 0;
    for (const char of code) {
        if (char === "\n") lines += 1;
    }
    return lines;
}

function createLineElement(code: string): HTMLPreElement {
    const lines = countLines(code) + 1;
    const maxLineWidth = lines.toString().length;
    let text = "";
    for (let i = 1; i < lines; ++i) {
        const node = i.toString().padStart(maxLineWidth);
        text += node;
        text += "\n";
    }
    const lineElement = document.createElement("pre");
    lineElement.classList.add("code-lines");
    lineElement.innerText = text;
    return lineElement;
}

async function checkStatus(): Promise<"running" | "done"> {
    const status = await data.status();

    if (status.running) {
        return "running";
    }
    const statusHtml = document.querySelector<HTMLSpanElement>(
        "#status",
    )!;
    statusHtml.innerText = "Done";
    document.body.classList.remove("status-waiting");
    document.body.classList.add("status-done");
    return "done";
}

function syntaxHighlight(code: string): string {
    const colors = {
        colorBackground: "#282828",
        colorForeground: "#fbf1c7",
        colorKeyword: "#fb4934",
        colorIdentifier: "#83a598",
        colorOperator: "#fe8019",
        colorSpecial: "#fe8019",
        colorType: "#fabd2f",
        colorBoolean: "#d3869b",
```

```
        colorNumber: "#d3869b",
        colorString: "#b8bb26",
        colorComment: "#928374",
        colorFunction: "#b8bb26",
        colorLineNumber: "#7c6f64",
    } as const;

    /*
    Keyword = { link = "GruvboxRed" },
    Identifier = { link = "GruvboxBlue" },
    Operator = { fg = colors.orange, italic =
config.italic.operators },
    Special = { link = "GruvboxOrange" },
    Type = { link = "GruvboxYellow" },
    Boolean = { link = "GruvboxPurple" },
    Number = { link = "GruvboxPurple" },
    String = { fg = colors.green, italic = config.italic.strings },
    Comment = { fg = colors.gray, italic = config.italic.comments
},
    Function = { link = "GruvboxGreenBold" },
    */

    let matches: {
        index: number;
        length: number;
        color: string;
        extra: string;
    }[] = [];

    function addMatches(color: string, re: RegExp, extra = "") {
        for (const match of code.matchAll(re)) {
            matches.push({
                index: match.index,
                length: match[1].length,
                color,
                extra,
            });
        }
    }
    function addKeywordMatches(color: string, keywords: string[]) {
        addMatches(
            color,
            new RegExp(
                `(?<!\\w)(${
                    keywords.map((kw) => `(?:${kw})`).join("|")
                })(?!\\w)`,
                "g",
            ),
        );
    }

    for (let i = 0; i < code.length; ++i) {
        if (code[i] !== '"') {
```

```
            continue;
        }
        let last = code[i];
        const index = i;
        i += 1;
        while (i < code.length && !(code[i] === '"' && last !== "\
\")) {
            last = code[i];
            i += 1;
        }
        if (i < code.length) {
            i += 1;
        }
        matches.push({
            index,
            length: i - index,
            color: colors.colorString,
            extra: "font-style: italic;",
        });
    }

    {
        let last = "";
        for (let i = 0; i < code.length; ++i) {
            if (last === "/" && code[i] === "/") {
                const index = i - 1;
                while (i < code.length && code[i] !== "\n") {
                    i += 1;
                }
                matches.push({
                    index,
                    length: i - index,
                    color: colors.colorComment,
                    extra: "font-style: italic;",
                });
            }
            last = code[i];
        }
    }

    addKeywordMatches(
        colors.colorKeyword,
        [
            "break",
            "return",
            "let",
            "fn",
            "if",
            "else",
            "struct",
            "import",
            "or",
            "and",
```

```
            "not",
            "loop",
            "while",
            "for",
            "in",
        ],
    );
    addKeywordMatches(colors.colorSpecial, ["null"]);
    addKeywordMatches(colors.colorType, ["int", "string", "bool"]);
    addKeywordMatches(colors.colorBoolean, ["false", "true"]);
    addMatches(
        colors.colorOperator,
        new RegExp(
            `(${
                [
                    "\\+=",
                    "\\-=",
                    "\\+",
                    "\\->",
                    "\\-",
                    "\\*",
                    "/",
                    "==",
                    "!=",
                    "<=",
                    ">=",
                    "=",
                    "<",
                    ">",
                    "\\.",
                    "::<",
                    "::",
                    ":",
                ].map((kw) => `(?:${kw})`).join("|")
            })`,
            "g",
        ),
    );
    addMatches(
        colors.colorNumber,
        /(0|(?:[1-9][0-9]*)|(?:0[0-7]+)|(?:0x[0-9a-fA-F]+)|
(?:0b[01]+))/g,
    );
    addMatches(
        colors.colorFunction,
        /([a-zA-Z_]\w*(?=\())/g,
        "font-weight: 700;",
    );
    addMatches(colors.colorIdentifier, /([a-z_]\w*)/g);
    addMatches(colors.colorType, /([A-Z_]\w*)/g);

    matches = matches.reduce<typeof matches>(
        (acc, match) =>
```

```
            acc.find((m) => m.index === match.index) === undefined
                ? (acc.push(match), acc)
                : acc,
        [],
    );
    matches.sort((a, b) => a.index - b.index);

    let highlighted = "";
    let i = 0;
    for (const match of matches) {
        if (match.index < i) {
            continue;
        }
        while (i < match.index) {
            highlighted += code[i];
            i += 1;
        }
        const section = code.slice(match.index, match.index +
match.length);
        highlighted +=
            `<span style="color: ${match.color};
${match.extra}">${section}</span>`;
        i += section.length;
    }
    while (i < code.length) {
        highlighted += code[i];
        i += 1;
    }

    return highlighted;
}

function sourceCode(view: Element, codeData: string) {
    const outerContainer = document.createElement("div");
    outerContainer.classList.add("code-container");

    const innerContainer = document.createElement("div");
    innerContainer.classList.add("code-container-inner");

    const lines = createLineElement(codeData);
    const code = document.createElement("pre");

    code.classList.add("code-source");
    code.innerHTML = syntaxHighlight(codeData);
    innerContainer.append(lines, code);
    outerContainer.append(innerContainer);
    view.replaceChildren(outerContainer);
}

function createRadio(
    id: string,
    content: string,
    checked: boolean,
```

```
): [HTMLDivElement, HTMLInputElement] {
    const label = document.createElement("label");
    label.htmlFor = id;
    label.innerText = content;
    const input = document.createElement("input");
    input.id = id;
    input.name = "coverage-radio";
    input.type = "radio";
    input.hidden = true;
    input.checked = checked;
    const container = document.createElement("div");
    container.classList.add("coverage-radio-group");
    container.append(input, label);
    return [container, input];
}

async function codeCoverage(view: Element, codeData: string) {
    const codeCoverageData = await data.codeCoverageData();

    const outerContainer = document.createElement("div");
    outerContainer.classList.add("code-container");
    outerContainer.classList.add("code-coverage");

    const innerContainer = document.createElement("div");
    innerContainer.classList.add("code-container-inner");

    const [perfGroup, perfInput] = createRadio(
        "performance-coverage",
        "Performance view",
        true,
    );
    const [testGroup, testInput] = createRadio(
        "test-coverage",
        "Test view",
        false,
    );

    function load(mode: CodeCovRender, tooltip: HTMLDivElement) {
        const lines = createLineElement(codeData);
        const code = loadCodeCoverage(
            codeData,
            codeCoverageData,
            tooltip,
            mode,
        );
        innerContainer.replaceChildren(lines, code);
    }

    const radios = document.createElement("div");
    radios.append(perfGroup, testGroup);
    radios.classList.add("coverage-radio");
    const tooltip = document.createElement("div");
    tooltip.id = "covers-tooltip";
```

```
    tooltip.hidden = true;
    outerContainer.append(innerContainer);
    view.replaceChildren(outerContainer, tooltip, radios);

    if (perfInput.checked) {
        load("performance", tooltip);
    } else if (testInput.checked) {
        load("test-coverage", tooltip);
    }

    perfInput.addEventListener("input", () => load("performance",
tooltip));
    testInput.addEventListener("input", () => load("test-coverage",
tooltip));
}

async function main() {
    type RenderFns = {
        "source-code": () => void;
        "code-coverage": () => void;
        "flame-graph": () => void;
    };

    const codeData = await data.codeData();

    const view = document.querySelector("#view")!;
    const renderFunctions: RenderFns = {
        "source-code": () => sourceCode(view, codeData),
        "code-coverage": async () => await codeCoverage(view,
codeData),
        "flame-graph": async () => {
            const flameGraphData = await data.flameGraphData();
            const flameGraphFnNames = await
data.flameGraphFnNames();

            const container = document.createElement("div");
            container.classList.add("flame-graph");
            container.id = "flame-graph";
            const view = document.querySelector("#view")!;
            view.replaceChildren(container);
            loadFlameGraph(flameGraphData, flameGraphFnNames,
container);
        },
    };

    const viewRadios: NodeListOf<HTMLInputElement> =
document.querySelectorAll(
        'input[name="views"]',
    );
    for (const input of viewRadios) {
        input.addEventListener("input", (ev) => {
            const target = ev.target as HTMLInputElement;
            const value = target.value as keyof RenderFns;
```

```
            renderFunctions[value]();
        });
        if (input.checked) {
            const value = input.value as keyof RenderFns;
            renderFunctions[value]();
        }
    }

    checkStatus().then((status) => {
        if (status == "done") {
            return;
        }
        const interval = setInterval(async () => {
            const status = await checkStatus();
            if (status == "done") {
                clearInterval(interval);
            }
        }, 500);
    });
}

main();
```

web/public/src/code_coverage.ts

```
import * as data from "./data.ts";

type Color = { r: number; g: number; b: number };

function lerp2(ratio: number, start: number, end: number) {
    return (1 - ratio) * start + ratio * end;
}

function lerp3(ratio: number, start: number, middle: number, end:
number) {
    return (1 - ratio) * lerp2(ratio, start, middle) +
        ratio * lerp2(ratio, middle, end);
}

function colorLerp(
    ratio: number,
    start: Color,
    middle: Color,
    end: Color,
): Color {
    return {
        r: lerp3(ratio, start.r, middle.r, end.r),
        g: lerp3(ratio, start.g, middle.g, end.g),
        b: lerp3(ratio, start.b, middle.b, end.b),
    };
}

function colorToString(color: Color): string {
```

```
    return `rgb(${color.r}, ${color.g}, ${color.b})`;
}

const GREEN = { r: 42, g: 121, b: 82 };
const YELLOW = {
    r: 247,
    g: 203,
    b: 21,
};
const RED = {
    r: 167,
    g: 29,
    b: 49,
};

export type CodeCovRender = "performance" | "test-coverage";

export function loadCodeCoverage(
    text: string,
    input: data.CodeCovEntry[],
    tooltip: HTMLElement,
    mode: CodeCovRender,
): HTMLPreElement {
    const container = document.createElement("pre");
    container.classList.add("code-source");
    if (input.length === 0) {
        return container;
    }
    const entries = input.toSorted((
        a: data.CodeCovEntry,
        b: data.CodeCovEntry,
    ) => b.index - a.index);
    const charEntries: { [key: string]: data.CodeCovEntry } = {};
    const elements: HTMLElement[] = [];
    let line = 1;
    let col = 1;
    const maxPerfCovers = entries.map((v) => v.covers).reduce((acc,
v) =>
        acc > Math.log10(v) ? acc : Math.log10(v)
    );
    for (let index = 0; index < text.length; ++index) {
        if (text[index] === "\n") {
            col = 1;
            line += 1;
            const newlineSpan = document.createElement("span");
            newlineSpan.innerText = "\n";
            elements.push(newlineSpan);
            continue;
        }
        const entry = entries.find((entry) => index >=
entry.index);
        if (!entry) {
            throw new Error("unreachable");
```

```
        }
        charEntries[`${line}-${col}`] = entry;

        const perfColor = (ratio: number) => {
            const clr = colorLerp(ratio, GREEN, YELLOW, RED);
            return colorToString(clr);
        };

        const span = document.createElement("span");
        span.style.backgroundColor = mode === "performance"
            ? perfColor(
                Math.log10(entry.covers) / maxPerfCovers,
            )
            : entry.covers > 0
            ? colorToString(GREEN)
            : colorToString(RED);
        span.innerText = text[index];
        span.dataset.covers = entry.covers.toString();
        elements.push(span);
        col += 1;
    }
    container.append(...elements);
    document.addEventListener("mousemove", (event) => {
        const [x, y] = [event.clientX, event.clientY];
        const outerBox = container.getBoundingClientRect();
        if (!positionInBox([x, y], outerBox)) {
            tooltip.hidden = true;
            return;
        }
        const element = elements.find((element) => {
            if (typeof element === "string") {
                return false;
            }
            if (!element.dataset.covers) {
                return false;
            }
            const isIn = positionInBox([x, y],
element.getBoundingClientRect());
            return isIn;
        });
        if (!element) {
            tooltip.hidden = true;
            return;
        }
        const maybeCovers = element.dataset.covers;
        if (!maybeCovers) {
            throw new Error("unreachable");
        }
        const covers = parseInt(maybeCovers);
        tooltip.hidden = false;
        tooltip.style.left = `${event.clientX + 20}px`;
        tooltip.style.top = `${event.clientY + 20}px`;
        tooltip.innerText = `Ran ${covers} time${covers !== 1 ? "s"
```

```
: ""}`;
    });
    return container;
}

function positionInBox(
    position: [number, number],
    boundingRect: {
        left: number;
        top: number;
        right: number;
        bottom: number;
    },
) {
    const [x, y] = position;
    const outside = x < boundingRect.left ||
        x >= boundingRect.right || y < boundingRect.top ||
        y >= boundingRect.bottom;
    return !outside;
}
```

web/runtime.ts

```
export class Runtime {
    constructor(private port: number) {}

    async connect(): Promise<RuntimeConnection> {
        return new RuntimeConnection(
            await Deno.connect({
                port: this.port,
            }),
        );
    }
}

export class RuntimeConnection {
    constructor(private connection: Deno.Conn) {}

    async write(text: string): Promise<void> {
        const req = new TextEncoder().encode(text);
        await this.connection.write(req);
    }

    async send<T>(value: T): Promise<void> {
        await this.write(JSON.stringify(value));
    }

    async read(): Promise<string> {
        let result = "";
        while (true) {
            const buf = new Uint8Array(256);
            const readRes = await this.connection.read(buf);
            if (readRes != null) {
```

```
                result += new TextDecoder().decode(buf.slice(0,
readRes));
            } else {
                break;
            }
        }
        return result;
    }

    async receive<T>(): Promise<T> {
        return JSON.parse(await this.read()) as T;
    }

    close() {
        this.connection.close();
    }
}
```

web/main.ts

```
import { Application, Router } from "jsr:@oak/oak";
import { parseArgs } from "jsr:@std/cli/parse-args";
import { Runtime } from "./runtime.ts";
import * as compiler from "../compiler/mod.ts";

const port = 8000;

const flags = parseArgs(Deno.args, {
    boolean: ["flame-graph", "code-coverage"],
});

if (flags._.length !== 1) {
    throw new Error("please specify a filename");
}

//

const filepath = flags._[0] as string;
const text = await Deno.readTextFile(filepath);

const runtime = new Runtime(13370);

async function runProgramWithDebug(program: number[]) {
    const connection = await runtime.connect();
    connection.send({
        type: "run-debug",
        program,
    });
    const res = await connection.receive<{
        ok: boolean;
    }>();
    connection.close();
    if (!res.ok) {
```

```
        throw new Error("could not run code");
    }
}

const { program, fnNames } = await
compiler.compileWithDebug(filepath);
await runProgramWithDebug(program);

//

const router = new Router();

router.get("/api/source", (ctx) => {
    ctx.response.body = { ok: true, filepath, text };
    ctx.response.status = 200;
    ctx.respond = true;
});

router.get("/api/status", async (ctx) => {
    const connection = await runtime.connect();
    connection.send({ type: "status" });
    const res = await connection.receive<{
        ok: boolean;
        status: { running: boolean };
    }>();
    connection.close();
    if (!res.ok) {
        ctx.response.body = { ok: false };
        ctx.response.status = 500;
        ctx.respond = true;
        return;
    }
    ctx.response.body = { ok: true, status: res.status };
    ctx.response.status = 200;
    ctx.respond = true;
});

router.get("/api/flame-graph", async (ctx) => {
    const connection = await runtime.connect();
    connection.send({ type: "flame-graph" });
    const res = await connection.receive<{
        ok: boolean;
        flameGraph: string;
    }>();
    connection.close();
    if (!res.ok) {
        ctx.response.body = { ok: false };
        ctx.response.status = 500;
        ctx.respond = true;
        return;
    }
    ctx.response.body = { ok: true, flameGraph: res.flameGraph };
    ctx.response.status = 200;
```

```
    ctx.respond = true;
});

router.get("/api/flame-graph-fn-names", (ctx) => {
    ctx.response.body = { ok: true, fnNames };
    ctx.response.status = 200;
    ctx.respond = true;
});

router.get("/api/code-coverage", async (ctx) => {
    const connection = await runtime.connect();
    connection.send({ type: "code-coverage" });
    const res = await connection.receive<{
        ok: boolean;
        codeCoverage: string;
    }>();
    connection.close();
    if (!res.ok) {
        ctx.response.body = { ok: false };
        ctx.response.status = 500;
        ctx.respond = true;
        return;
    }
    ctx.response.body = { ok: true, codeCoverage: res.codeCoverage
};
    ctx.response.status = 200;
    ctx.respond = true;
});

//

const app = new Application();
app.use(router.routes());
app.use(router.allowedMethods());
app.use(async (ctx, next) => {
    try {
        await ctx.send({ root: "./public", index: "index.html" });
    } catch {
        next();
    }
});
const listener = app.listen({ port });
console.log(`Devtools at http://localhost:${port}/`);
await listener;
```

slige-build-run-all.sh

```
#!/bin/bash

set -e

FILE_FULL_PATH=$(readlink -f $1)
```

```
cd runtime
make
cd ..

cd web/public
deno task bundle
cd ../..

set +e
fuser -k 13370/tcp
set -e


./runtime/build/sliger &

cd web

deno run --allow-net --allow-read main.ts $FILE_FULL_PATH
```

runtime/to_json.cpp

```
#include "vm.hpp"
#include <string>

using namespace sliger;

void FlameGraphBuilder::to_json(json::Writer& writer) const
{
    fg_node_to_json(writer, 0);
}

void FlameGraphBuilder::fg_node_to_json(
    json::Writer& writer, size_t node_index) const
{
    const auto& node = this->nodes[node_index];
    writer << "{\"fn\":" << std::to_string(node.fn)
           << ",\"acc\":" << std::to_string(node.acc)
           << ",\"parent\":" << std::to_string(node.parent)
           << ",\"children\":[";
    auto first = true;
    for (auto child_index : node.children) {
        if (!first) {
            writer << ",";
        }
        first = false;
        fg_node_to_json(writer, child_index);
    }
    writer << "]}";
}

void CodeCoverageBuilder::to_json(json::Writer& writer) const
{
```

```
    writer << "[";
    writer.add_comma_seperated(
        this->entries, [&](json::Writer& writer, CCPosEntry entry)
{
            writer << "{\"index\":" <<
std::to_string(entry.pos.index)
                   << ",\"line\":" <<
std::to_string(entry.pos.line)
                   << ",\"col\":" << std::to_string(entry.pos.col)
                   << ",\"covers\":" <<
std::to_string(entry.covers) << "}";
        });
    writer << "]";
}
```

runtime/rpc_server.cpp

```
#include "rpc_server.hpp"

extern "C" {
#include <arpa/inet.h>
#include <asm-generic/socket.h>
#include <netdb.h>
#include <netinet/in.h>
#include <stdlib.h>
#include <string>
#include <sys/socket.h>
#include <unistd.h>
}

using namespace sliger::rpc;

static auto create_address(uint16_t port) -> sockaddr_in
{
    return {
        .sin_family = AF_INET,
        .sin_port = ::htons(port),
        .sin_addr = { .s_addr = ::inet_addr("127.0.0.1") },
        .sin_zero = { 0 },
    };
}

auto Socket::init() -> Res<void>
{
    this->socket_fd = ::socket(AF_INET, SOCK_STREAM, 0);
    if (socket_fd < 0) {
        return Err { .msg
            = std::format("could not get socket ({})", socket_fd)
};
    };
    this->initialized = true;

    int trueValue = 1;
```

```
    ::setsockopt(
        this->socket_fd, SOL_SOCKET, SO_REUSEADDR, &trueValue,
sizeof(int));

    int err;

    auto address = create_address(13370);
    err = ::bind(socket_fd, (struct sockaddr*)&address,
sizeof(address));
    if (err < 0) {
        return Err { .msg = std::format("could not bind ({})", err)
};
    };

    err = ::listen(socket_fd, 0);
    if (err < 0) {
        return Err { .msg = std::format("could not listen ({})",
err) };
    }
    return {};
}

auto Socket::accept() -> Res<std::unique_ptr<Client>>
{
    auto client_address = create_address(13370);
    socklen_t address_size = sizeof(client_address);
    int client
        = ::accept(socket_fd, (struct sockaddr*)&client_address,
&address_size);
    if (client < 0) {
        return Err { .msg = std::format("could not accept ({})",
client) };
    }
    return std::make_unique<Client>(client);
}
auto Client::read(int8_t* buffer, size_t buffer_size) -> ssize_t
{
    return ::read(this->client_sock, buffer, buffer_size);
}
auto Client::write(uint8_t* buffer, size_t buffer_size) -> ssize_t
{
    return ::write(this->client_sock, buffer, buffer_size);
}

auto BufferedWriter::write(std::string message) -> Res<void>
{
    for (auto ch : message) {
        if (auto res = this->write((uint8_t)ch); !res.is_ok()) {
            return res.err();
        }
    }
    return {};
}
```

```cpp
auto BufferedWriter::write(uint8_t byte) -> Res<void>
{
    if (this->occupied >= length) {
        if (auto res = this->flush(); !res.is_ok()) {
            return res.err();
        }
    }
    this->buffer[this->occupied] = byte;
    this->occupied += 1;

    return {};
}

auto BufferedWriter::flush() -> Res<size_t>
{
    auto result = this->client->write(this->buffer, this-
>occupied);
    if (result < 0) {
        return Err("unable to write");
    }
    this->occupied = 0;
    return (size_t)result;
}
```

runtime/arch.hpp

```cpp
#pragma once

#include <cstdint>

namespace sliger {

// NOTICE: keep up to date with src/arch.ts

enum class Op : uint32_t {
    Nop = 0x00,
    PushNull = 0x01,
    PushInt = 0x02,
    PushBool = 0x03,
    PushString = 0x04,
    PushPtr = 0x05,
    Pop = 0x06,
    ReserveStatic = 0x07,
    LoadStatic = 0x08,
    StoreStatic = 0x09,
    LoadLocal = 0x0a,
    StoreLocal = 0x0b,
    Call = 0x0c,
    Return = 0x0d,
    Jump = 0x0e,
    JumpIfTrue = 0x0f,
    Builtin = 0x10,
```

```
    Duplicate = 0x11,
    Swap = 0x12,
    Add = 0x20,
    Subtract = 0x21,
    Multiply = 0x22,
    Divide = 0x23,
    Remainder = 0x24,
    Equal = 0x25,
    LessThan = 0x26,
    And = 0x27,
    Or = 0x28,
    Xor = 0x29,
    Not = 0x2a,
    SourceMap = 0x30,
};

enum class Builtin : uint32_t {
    IntToString = 0x00,
    StringConcat = 0x10,
    StringEqual = 0x11,
    StringCharAt = 0x12,
    StringLength = 0x13,
    StringPushChar = 0x14,
    StringToInt = 0x15,
    ArrayNew = 0x20,
    ArraySet = 0x21,
    ArrayPush = 0x22,
    ArrayAt = 0x23,
    ArrayLength = 0x24,
    StructSet = 0x30,
    Print = 0x40,
    FileOpen = 0x41,
    FileClose = 0x42,
    FileWriteString = 0x43,
    FileReadChar = 0x44,
    FileReadToString = 0x45,
    FileFlush = 0x46,
    FileEof = 0x47,
};

}
```

runtime/json.cpp

```
#include "json.hpp"
#include <cstdlib>
#include <format>
#include <memory>
#include <string>
#include <unordered_map>

using namespace sliger::json;
```

```
auto ident_tok_typs = std::unordered_map<std::string, TokTyp> {
    { "null", TokTyp::Null },
    { "false", TokTyp::False },
    { "true", TokTyp::True },
};

auto id_start_chars = "abcdefghijklmnopqrstuvwxyz"
                      "ABCDEFGHIJKLMNOPQRSTUVWXYZ_-";

auto id_tail_chars = "abcdefghijklmnopqrstuvwxyz"
                     "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
                     "1234567890-";

auto Lexer::next() -> Res<Tok>
{
    auto pos = this->pos();
    if (done()) {
        return Tok(TokTyp::Eof, pos);
    }
    if (test('"')) {
        step();
        auto value = std::string();
        while (!done() and !test('"')) {
            if (cur() == '\\') {
                step();
                if (done())
                    break;
                value.push_back([&] {
                    char ch = cur();
                    switch (ch) {
                        case 'n':
                            return '\n';
                        case 'r':
                            return '\r';
                        case 't':
                            return '\t';
                        case '0':
                            return '\0';
                        default:
                            return ch;
                    }
                }());
            } else {
                value.push_back(cur());
            }
            step();
        }
        if (!test('"')) {
            return Err {
                .pos = pos,
                .msg
                = std::format("malformed string, expected '\"', got
'{}' token",
```

```
                    this->cur()),
            };
        }
        step();
        return Tok(TokTyp::String, pos, intern_str(value));
    }
    auto step_n_ret = [&](auto tok) {
        step();
        return tok;
    };
    switch (cur()) {
        case '0':
            return step_n_ret(Tok(TokTyp::Float, pos, 0.0));
        case '{':
            return step_n_ret(Tok(TokTyp::LBrace, pos));
        case '}':
            return step_n_ret(Tok(TokTyp::RBrace, pos));
        case '[':
            return step_n_ret(Tok(TokTyp::LBracket, pos));
        case ']':
            return step_n_ret(Tok(TokTyp::RBracket, pos));
        case ',':
            return step_n_ret(Tok(TokTyp::Comma, pos));
        case ':':
            return step_n_ret(Tok(TokTyp::Colon, pos));
    }
    if (test_in(id_start_chars)) {
        auto value = std::string();
        while (test_in(id_tail_chars)) {
            value.push_back(cur());
            step();
        }
        if (ident_tok_typs.find(value) == ident_tok_typs.end()) {
            return Err {
                .pos = pos,
                .msg = std::format("unknown identifier \"{}\"",
value),
            };
        }
        return Tok(ident_tok_typs.at(value), pos);
    }
    if (test_in("123456789")) {
        auto value_str = std::string();
        while (test_in("1234567890")) {
            value_str.push_back(cur());
            step();
        }
        auto value = std::atof(value_str.c_str());
        return Tok(TokTyp::Float, pos, value);
    }
    auto ch = cur();
    step();
    return Err {
```

```
        .pos = pos,
        .msg = std::format("unknown character '{}'", ch),
    };
}

auto Parser::parse_val() -> Res<std::unique_ptr<Value>>
{
    if (not this->cur.ok())
        return this->cur.err();
    auto cur = this->cur.val();
    switch (cur.typ) {
        case TokTyp::Eof:
            return Err {
                .pos = cur.pos,
                .msg = "expected value, got eof",
            };
        case TokTyp::String: {
            auto value = this->lexer.val(cur.val_id);
            step();
            return
Res<std::unique_ptr<Value>>(std::make_unique<String>(value));
        }
        case TokTyp::Float: {
            auto value = cur.float_val;
            step();
            return
Res<std::unique_ptr<Value>>(std::make_unique<Number>(value));
        }
        case TokTyp::False: {
            step();
            return
Res<std::unique_ptr<Value>>(std::make_unique<Bool>(false));
        }
        case TokTyp::True: {
            step();
            return
Res<std::unique_ptr<Value>>(std::make_unique<Bool>(true));
        }
        case TokTyp::Null: {
            step();
            return
Res<std::unique_ptr<Value>>(std::make_unique<Null>());
        }
        case TokTyp::LBrace: {
            step();
            ObjectFields fields;
            if (curtyp() != TokTyp::RBrace) {
                if (curtyp() != TokTyp::String) {
                    return unexpected_tok_err(
                        TokTyp::String, "malformed object");
                }
                auto key = this->lexer.val(this->cur.val().val_id);
                step();
```

```
                if (curtyp() != TokTyp::Colon) {
                    return unexpected_tok_err(
                        TokTyp::Colon, "malformed object");
                }
                step();
                auto value = parse_val();
                if (not value.ok()) {
                    return value.err();
                }
                fields.insert_or_assign(key,
std::move(value.val()));
                while (curtyp() == TokTyp::Comma) {
                    step();
                    if (curtyp() != TokTyp::String) {
                        return unexpected_tok_err(
                            TokTyp::String, "malformed object");
                    }
                    auto key = this->lexer.val(this-
>cur.val().val_id);
                    step();
                    if (curtyp() != TokTyp::Colon) {
                        return unexpected_tok_err(
                            TokTyp::Colon, "malformed object");
                    }
                    step();
                    auto value = parse_val();
                    if (not value.ok()) {
                        return value.err();
                    }
                    fields.insert_or_assign(key,
std::move(value.val()));
                }
            }
            if (curtyp() != TokTyp::RBrace) {
                return unexpected_tok_err(TokTyp::RBrace,
"malformed object");
            }
            step();
            return Res<std::unique_ptr<Value>>(
                std::make_unique<Object>(std::move(fields)));
        }
        case TokTyp::LBracket: {
            step();
            ArrayValues values;
            if (curtyp() != TokTyp::RBracket) {
                auto value = parse_val();
                if (not value.ok()) {
                    return value.err();
                }
                values.push_back(std::move(value.val()));
                while (curtyp() == TokTyp::Comma) {
                    step();
                    auto value = parse_val();
```

```
                    if (not value.ok()) {
                        return value.err();
                    }
                    values.push_back(std::move(value.val()));
                }
            }
            if (curtyp() != TokTyp::RBracket) {
                return unexpected_tok_err(TokTyp::RBracket,
"malformed array");
            }
            step();
            return Res<std::unique_ptr<Value>>(
                std::make_unique<Array>(std::move(values)));
        }
        case TokTyp::RBrace:
        case TokTyp::RBracket:
        case TokTyp::Comma:
        case TokTyp::Colon:
            return Err {
                .pos = cur.pos,
                .msg = std::format("expected value, got '{}'
token",
                    tok_typ_to_string(cur.typ)),
            };
            break;
    }
    return Err {
        .pos = cur.pos,
        .msg = std::format(
            "internal error, could not parse '{}'",
tok_typ_to_string(cur.typ)),
    };
}

auto Parser::unexpected_tok_err(TokTyp expected, std::string_view
msg)
    -> Res<std::unique_ptr<Value>>
{
    return Err {
        .pos = this->cur.val().pos,
        .msg = std::format("{}, expected '{}', got '{}'", msg,
            tok_typ_to_string(expected),
            tok_typ_to_string(this->cur.val().typ)),
    };
}
```

runtime/to_string.cpp

```
#include "json.hpp"
#include "vm.hpp"
#include <string>

using namespace sliger;
```

```cpp
auto sliger::maybe_op_to_string(uint32_t value) -> std::string
{
    switch (static_cast<Op>(value)) {
            /* clang-format off */
        case Op::Nop: return "Nop";
        case Op::PushNull: return "PushNull";
        case Op::PushInt: return "PushInt";
        case Op::PushBool: return "PushBool";
        case Op::PushString: return "PushString";
        case Op::PushPtr: return "PushPtr";
        case Op::Pop: return "Pop";
        case Op::ReserveStatic: return "ReserveStatic";
        case Op::LoadStatic: return "LoadStatic";
        case Op::StoreStatic: return "StoreStatic";
        case Op::LoadLocal: return "LoadLocal";
        case Op::StoreLocal: return "StoreLocal";
        case Op::Call: return "Call";
        case Op::Return: return "Return";
        case Op::Jump: return "Jump";
        case Op::JumpIfTrue: return "JumpIfTrue";
        case Op::Builtin: return "Builtin";
        case Op::Duplicate: return "Duplicate";
        case Op::Swap: return "Swap";
        case Op::Add: return "Add";
        case Op::Subtract: return "Subtract";
        case Op::Multiply: return "Multiply";
        case Op::Divide: return "Divide";
        case Op::Remainder: return "Remainder";
        case Op::Equal: return "Equal";
        case Op::LessThan: return "LessThan";
        case Op::And: return "And";
        case Op::Or: return "Or";
        case Op::Xor: return "Xor";
        case Op::Not: return "Not";
        case Op::SourceMap: return "SourceMap";
            /* clang-format on */
    }
    return std::to_string(value);
}

auto sliger::maybe_builtin_to_string(uint32_t value) -> std::string
{
    switch (static_cast<Builtin>(value)) {
            /* clang-format off */
        case Builtin::IntToString: return "IntToString";
        case Builtin::StringConcat: return "StringConcat";
        case Builtin::StringEqual: return "StringEqual";
        case Builtin::StringCharAt: return "StringCharAt";
        case Builtin::StringLength: return "StringLength";
        case Builtin::StringPushChar: return "StringPushChar";
        case Builtin::StringToInt: return "StringToInt";
        case Builtin::ArrayNew: return "ArrayNew";
```

```
        case Builtin::ArraySet: return "ArraySet";
        case Builtin::ArrayPush: return "ArrayPush";
        case Builtin::ArrayAt: return "ArrayAt";
        case Builtin::ArrayLength: return "ArrayLength";
        case Builtin::StructSet: return "StructSet";
        case Builtin::Print: return "Print";
        case Builtin::FileOpen: return "FileOpen";
        case Builtin::FileClose: return "FileClose";
        case Builtin::FileWriteString: return "FileWrite";
        case Builtin::FileReadToString: return "FileReadToString";
        case Builtin::FileFlush: return "FileFlush";
        case Builtin::FileEof: return "FileEof";
        /* clang-format on */
        default:
            return std::to_string(value);
    }
}

auto json::tok_typ_to_string(json::TokTyp typ) -> std::string
{
    using namespace json;

    switch (typ) {
            /* clang-format off */
        case TokTyp::Eof: return "Eof";
        case TokTyp::String: return "String";
        case TokTyp::Float: return "Float";
        case TokTyp::False: return "False";
        case TokTyp::True: return "True";
        case TokTyp::Null: return "Null";
        case TokTyp::LBrace: return "LBrace";
        case TokTyp::RBrace: return "RBrace";
        case TokTyp::LBracket: return "LBracket";
        case TokTyp::RBracket: return "RBracket";
        case TokTyp::Comma: return "Comma";
        case TokTyp::Colon: return "Colon";
            /* clang-format on */
    }
    std::unreachable();
}
```

runtime/alloc.hpp

```
#pragma once

#include "value.hpp"
#include <cstddef>
#include <cstdint>
#include <optional>
#include <string>
#include <unordered_map>
#include <variant>
#include <vector>
```

```
namespace sliger::heap {

struct Array {
    std::vector<Value> values;

    auto at(int32_t index) & -> Value&;
};

struct Struct {
    std::unordered_map<std::string, Value> fields;
};

enum class AllocType {
    Value,
    Array,
    Struct,
};

// clang-format off
template <AllocType type> struct AllocTypeType {};
template <> struct AllocTypeType<AllocType::Value> { using Type =
Value; };
template <> struct AllocTypeType<AllocType::Array> { using Type =
Array; };
template <> struct AllocTypeType<AllocType::Struct> { using Type =
Struct; };
// clang-format on

struct AllocItem {
    AllocItem(Value&& val)
        : type(AllocType::Value)
        , value(val)
    {
    }
    AllocItem(Array&& val)
        : type(AllocType::Array)
        , value(val)
    {
    }
    AllocItem(Struct&& val)
        : type(AllocType::Struct)
        , value(val)
    {
    }

    template <AllocType AT> inline auto as() & ->
AllocTypeType<AT>::Type&
    {
        return std::get<typename AllocTypeType<AT>::Type>(this-
>value);
    }
    template <AllocType AT>
```

```
    inline auto as() const& -> const AllocTypeType<AT>::Type&
    {
        return std::get<typename AllocTypeType<AT>::Type>(this-
>value);
    }
    template <AllocType AT> inline auto as() && ->
AllocTypeType<AT>::Type&&
    {
        return std::move(
            std::get<typename AllocTypeType<AT>::Type>(this-
>value));
    }
    template <AllocType AT>
    inline auto as() const&& -> const AllocTypeType<AT>::Type&&
    {
        return std::move(
            std::get<typename AllocTypeType<AT>::Type>(this-
>value));
    }

    inline auto as_value() & -> Value& { return
std::get<Value>(this->value); }
    inline auto as_value() const& -> const Value&
    {
        return std::get<Value>(this->value);
    }
    inline auto as_array() & -> Array& { return
std::get<Array>(this->value); }
    inline auto as_array() const& -> const Array&
    {
        return std::get<Array>(this->value);
    }
    inline auto as_struct() & -> Struct&
    {
        return std::get<Struct>(this->value);
    }
    inline auto as_struct() const& -> const Struct&
    {
        return std::get<Struct>(this->value);
    }

    AllocType type;
    std::variant<Value, Array, Struct> value;
};

enum class ErrType {
    InvalidPtrAccess,
    CannotAllocate,
};

template <typename T> struct Res {
    Res(T val)
        : m_val(std::forward<T>(val))
```

```
    {
    }
    Res(ErrType err)
        : m_err(err)
    {
    }

    auto ok() const -> bool { return m_val.has_value(); }
    auto val() & -> T& { return m_val.value(); }
    auto val() const& -> const T& { return m_val.value(); }
    auto val() && -> T&& { return std::move(m_val.value()); }
    auto val() const&& -> const T&& { return
std::move(m_val.value()); }
    auto err() -> ErrType { return m_err.value(); }

    std::optional<T> m_val;
    std::optional<ErrType> m_err;
};

class Heap {
public:
    inline auto can_allocate() const -> bool
    {
        return this->sel->size() < this->max_size;
    }

    inline void collect(const std::vector<uint32_t>&
ptr_stack_values)
    {
        this->other->reserve(this->max_size);
        for (auto ptr : ptr_stack_values) {
            move_item_to_other(ptr);
        }
        this->sel->clear();
        std::swap(this->sel, this->other);
        if (this->sel->size() + 1 >= this->max_size) {
            this->max_size *= 2;
            this->sel->reserve(this->max_size);
        } else if (this->sel->size() * 2 < this->max_size) {
            this->max_size /= 2;
            this->sel->shrink_to_fit();
            this->sel->reserve(this->max_size);
        }
    }

    inline auto at(uint32_t ptr) -> Res<AllocItem*>
    {
        if (ptr >= this->sel->size()) {
            return ErrType::InvalidPtrAccess;
        }
        return &this->sel->at(ptr);
    }
```

```
    template <AllocType type> auto alloc() -> Res<uint32_t>
    {
        if (not can_allocate()) {
            return ErrType::CannotAllocate;
        }
        auto ptr = static_cast<uint32_t>(this->sel->size());
        this->sel->push_back(typename AllocTypeType<type>::Type
{});
        return ptr;
    }

private:
    inline void move_item_to_other(uint32_t ptr)
    {
        auto res = at(ptr);
        if (!res.ok())
            return;
        auto val = res.val();
        switch (val->type) {
            case AllocType::Value: {
                auto& v = val->as<AllocType::Value>();
                if (v.type() == ValueType::Ptr) {
                    move_item_to_other(v.template
as<ValueType::Ptr>().value);
                }
                break;
            }
            case AllocType::Array: {
                auto& vs = val->as<AllocType::Array>();
                for (auto& v : vs.values) {
                    if (v.type() == ValueType::Ptr) {
                        move_item_to_other(
                            v.template as<ValueType::Ptr>().value);
                    }
                }
                break;
            }
            case AllocType::Struct: {
                auto& vs = val->as<AllocType::Struct>();
                for (auto& [key, v] : vs.fields) {
                    if (v.type() == ValueType::Ptr) {
                        move_item_to_other(
                            v.template as<ValueType::Ptr>().value);
                    }
                }
                break;
            }
        }
    }

    size_t max_size = 4;

    std::vector<AllocItem> heap_1;
```

```
    std::vector<AllocItem> heap_2;

    std::vector<AllocItem>* sel = &heap_1;
    std::vector<AllocItem>* other = &heap_2;
};

}
```

runtime/instruction_size.cpp

```
#include "arch.hpp"
#include "vm.hpp"

using namespace sliger;

size_t VM::instruction_size(size_t i) const
{
    switch (static_cast<Op>(this->program.at(i))) {
        case Op::Nop:
        case Op::PushNull:
            return 1;
        case Op::PushInt:
        case Op::PushBool:
            return 2;
        case Op::PushString: {
            auto string_length = this->program.at(i + 1);
            return 2 + string_length;
        }
        case Op::PushPtr:
            return 2;
        case Op::Pop:
            return 1;
        case Op::ReserveStatic:
        case Op::LoadStatic:
        case Op::StoreStatic:
        case Op::LoadLocal:
        case Op::StoreLocal:
        case Op::Call:
            return 2;
        case Op::Return:
        case Op::Jump:
        case Op::JumpIfTrue:
            return 1;
        case Op::Builtin:
            return 2;
        case Op::Duplicate:
        case Op::Swap:
        case Op::Add:
        case Op::Subtract:
        case Op::Multiply:
        case Op::Divide:
        case Op::Remainder:
        case Op::Equal:
```

```
        case Op::LessThan:
        case Op::And:
        case Op::Or:
        case Op::Xor:
        case Op::Not:
            return 1;
        case Op::SourceMap:
            return 4;
    }
    return 1;
}
```

runtime/vm.hpp

```
#pragma once

#include "alloc.hpp"
#include "arch.hpp"
#include "json.hpp"
#include "value.hpp"
#include <cstddef>
#include <cstdint>
#include <cstdio>
#include <string>
#include <utility>
#include <vector>

namespace sliger {

struct SourcePos {
    int index;
    int line;
    int col;
};

struct FGNode {
    FGNode(uint32_t fn, size_t parent)
        : fn(fn)
        , parent(parent)
    {
    }

    uint32_t fn;
    int64_t acc = 0;
    int64_t ic_start = 0;
    size_t parent;
    // the vector's data may be placed all over the heap. this
really really
    // sucks. expect cachemisses when calling infrequently called
functions. we
    // might be lucky, that the current function and frequent call
paths will be
    // cached, but we have no way to assure this.
```

```
    //
    // maybe to fix this, a many-to-many relation table, using one
single
    // vector could be used. that would reduce cache misses, in
exchange for
    // table lookups.
    std::vector<size_t> children = {};
};

class FlameGraphBuilder : public json::ToAndFromJson {
public:
    inline void report_call(uint32_t fn, int64_t ic_start)
    {
        size_t found = find_or_create_child(fn);
        this->nodes[found].ic_start = ic_start;
        this->current = found;
    }

    inline void report_return(int64_t ic_end)
    {
        int64_t diff = ic_end - this->nodes[this-
>current].ic_start;
        this->nodes[this->current].acc += diff;
        this->current = this->nodes[this->current].parent;
    }

    inline void calculate_midway_result(int64_t ic)
    {
        calculate_node_midway_result(ic, this->current);
    }

    void to_json(json::Writer& writer) const override;

private:
    inline auto find_or_create_child(uint32_t fn) -> size_t
    {
        auto found_child_index = this->find_child(fn);
        if (found_child_index.has_value())
            return found_child_index.value();
        size_t new_child_index = this->nodes.size();
        this->nodes.push_back(FGNode(fn, this->current));
        this->nodes[this-
>current].children.push_back(new_child_index);
        return new_child_index;
    }

    inline auto find_child(uint32_t fn) const ->
std::optional<size_t>
    {
        for (auto child_idx : this->nodes[this->current].children)
{
            if (fn == this->nodes[child_idx].fn) {
                return child_idx;
```

```
            }
        }
        return {};
    }

    inline void calculate_node_midway_result(int64_t ic, size_t
node_index)
    {
        int64_t diff = ic - this->nodes[node_index].ic_start;
        this->nodes[node_index].acc += diff;
        this->nodes[node_index].ic_start = ic;
        if (node_index == 0)
            return;
        calculate_node_midway_result(ic, this-
>nodes[node_index].parent);
    }

    void fg_node_to_json(json::Writer& writer, size_t node_index)
const;

    std::vector<FGNode> nodes = { FGNode(0, 0) };
    size_t current = 0;
};

struct CCPosEntry {
    SourcePos pos;
    int64_t covers = 0;
};

class CodeCoverageBuilder : public json::ToAndFromJson {
public:
    inline void make_sure_entry_exists(SourcePos pos)
    {
        find_or_create_entry(pos);
    }

    /// call when leaving a source location
    inline void report_cover(SourcePos pos)
    {
        size_t entry_index = find_or_create_entry(pos);
        this->entries[entry_index].covers += 1;
    }

    void to_json(json::Writer& writer) const override;

private:
    inline size_t find_or_create_entry(SourcePos pos)
    {
        if (auto found_index = find_pos_entry(pos);
found_index.has_value())
            return found_index.value();
        size_t new_index = this->entries.size();
        this->entries.push_back({ .pos = pos });
```

```
        return new_index;
    }

    inline std::optional<size_t> find_pos_entry(SourcePos pos)
const
    {
        for (size_t i = 0; i < this->entries.size(); ++i)
            if (this->entries[i].pos.index == pos.index)
                return i;
        return {};
    }

    std::vector<CCPosEntry> entries = {};
};

struct VMOpts {
    bool flame_graph;
    bool code_coverage;
    bool print_debug;
};

class VM {
public:
    VM(std::vector<uint32_t> program, VMOpts opts)
        : opts(opts)
        , program(std::move(program))
    {
    }

    void run_until_done();
    void run_n_instructions(size_t amount);
    void run_instruction();

    inline auto done() const -> bool
    {
        return this->pc >= this->program.size();
    }

    inline auto flame_graph_json() const -> std::string
    {
        return json::to_json(this->flame_graph);
    }

    inline auto code_coverage_json() -> std::string
    {
        for (size_t i = 0; i < this->program.size(); ++i) {
            if (this->program.at(i) ==
std::to_underlying(Op::SourceMap)
                && this->program.size() - 1 - i >= 3) {
                auto index = static_cast<int32_t>(this-
>program.at(i + 1));
                auto line = static_cast<int32_t>(this->program.at(i
+ 2));
```

```
                auto col = static_cast<int32_t>(this->program.at(i
+ 3));
                this->code_coverage.make_sure_entry_exists(
                    { index, line, col });
            }
            i += instruction_size(i);
        }
        return json::to_json(this->code_coverage);
    }

    inline auto view_stack() const -> const std::vector<Value>&
    {
        return this->stack;
    }

    auto stack_repr_string(size_t max_items) const -> std::string;

private:
    void run_builtin(Builtin builtin_id);
    void run_string_builtin(Builtin builtin_id);
    void run_array_builtin(Builtin builtin_id);
    void run_file_builtin(Builtin builtin_id);

    inline void step() { this->pc += 1; }

    inline auto eat_op() -> Op
    {
        auto value = curr_as_op();
        step();
        return value;
    }
    inline auto curr_as_op() const -> Op
    {
        return static_cast<Op>(this->program[this->pc]);
    }

    inline auto eat_int32() -> int32_t
    {
        auto value = curr_as_int32();
        step();
        return value;
    }
    inline auto curr_as_int32() const -> int32_t
    {
        return static_cast<int32_t>(this->program[this->pc]);
    }

    inline auto eat_uint32() -> uint32_t
    {
        auto value = curr_as_uint32();
        step();
        return value;
    }
```

```
    inline auto curr_as_uint32() const -> uint32_t
    {
        return this->program[this->pc];
    }

    inline auto fn_stack_at(size_t idx) -> Value&
    {
        return this->stack.at(this->bp + idx);
    }
    void assert_program_has(size_t count);
    void assert_fn_stack_has(size_t count);
    void assert_stack_has(size_t count);
    inline void stack_push(Value&& value) { this-
>stack.push_back(value); }
    inline void stack_push(Value& value) { this-
>stack.push_back(value); }
    inline auto stack_pop() -> Value
    {
        auto value = this->stack.at(this->stack.size() - 1);
        this->stack.pop_back();
        return value;
    }
    size_t instruction_size(size_t i) const;

    VMOpts opts;
    uint32_t pc = 0;
    uint32_t bp = 0;
    std::vector<uint32_t> program;
    std::vector<Value> stack;
    std::vector<Value> statics;
    heap::Heap heap;
    SourcePos current_pos = { 0, 1, 1 };
    int64_t instruction_counter = 0;

    int32_t file_id_counter = 3;
    std::unordered_map<int32_t, FILE*> open_files {
        { 0, stdin },
        { 1, stdout },
        { 2, stderr },
    };

    FlameGraphBuilder flame_graph;
    CodeCoverageBuilder code_coverage;
};

auto maybe_op_to_string(uint32_t value) -> std::string;
auto maybe_builtin_to_string(uint32_t value) -> std::string;

}
```

runtime/vm_provider.hpp

```
#pragma once
```

```
#include "vm.hpp"
#include <cstdint>
#include <mutex>
#include <thread>
#include <vector>

namespace sliger::rpc {

class VmProvider {
public:
    VmProvider() { }

    VmProvider(const VmProvider&) = delete;
    VmProvider operator=(const VmProvider&) = delete;
    VmProvider(VmProvider&&) = delete;
    VmProvider operator=(VmProvider&&) = delete;

    auto load_and_start(std::vector<uint32_t> instructions) ->
void;
    auto flame_graph_json() -> std::optional<std::string>;
    auto code_coverage_json() -> std::optional<std::string>;

    auto done() -> bool;

private:
    void run_timeslot();

    std::mutex mutex;

    std::optional<VM> vm;
    std::optional<std::thread> running_thread;
};

}
```

runtime/actions.cpp

```
#include "actions.hpp"
#include "json.hpp"
#include "vm_provider.hpp"
#include <cstdlib>
#include <iostream>
#include <memory>

using namespace sliger::rpc::action;

auto Status::perform_action(
    std::unique_ptr<sliger::rpc::BufferedWriter> writer,
VmProvider& vm) -> void
{
    bool running = not vm.done();
```

```
    writer->write(std::format(
        "{{ \"ok\": true, \"status\": {{ \"running\": {} }} }}",
running));
    writer->flush();
};

auto FlameGraph::perform_action(
    std::unique_ptr<sliger::rpc::BufferedWriter> writer,
VmProvider& vm) -> void
{
    auto json = vm.flame_graph_json();
    if (json) {
        writer->write(std::format(
            "{{ \"ok\": true, \"flameGraph\": {}  }}",
json.value()));
    } else {
        writer->write("{ \"ok\": false }");
    }
    writer->flush();
};

auto CodeCoverage::perform_action(
    std::unique_ptr<sliger::rpc::BufferedWriter> writer,
VmProvider& vm) -> void
{
    auto json = vm.code_coverage_json();
    if (json) {
        writer->write(std::format(
            "{{ \"ok\": true, \"codeCoverage\": {} }}",
json.value()));
    } else {
        writer->write("{ \"ok\": false }");
    }
    writer->flush();
};

auto RunDebug::perform_action(
    std::unique_ptr<sliger::rpc::BufferedWriter> writer,
VmProvider& vm) -> void
{
    auto program = this->instructions;
    vm.load_and_start(program);
    writer->write("{ \"ok\": true }");
    writer->flush();
};

auto sliger::rpc::action::action_from_json(
    const sliger::json::Value& value) -> std::unique_ptr<Action>
{
    auto& obj = value.as<sliger::json::Object>();
    auto type = obj.fields.at("type")->as<sliger::json::String>();

    if (type.value == "status") {
```

```
        return std::make_unique<Status>();
    }

    if (type.value == "flame-graph") {
        return std::make_unique<FlameGraph>();
    }

    if (type.value == "code-coverage") {
        return std::make_unique<CodeCoverage>();
    }

    if (type.value == "run-debug") {
        sliger::json::ArrayValues values = std::move(
            obj.fields.at("program")-
>as<sliger::json::Array>().values);
        auto instructions = std::vector<uint32_t>();
        for (auto&& v : values) {
            auto value = v->as<sliger::json::Number>().value;
            instructions.push_back((uint32_t)value);
        }
        return std::make_unique<RunDebug>(instructions);
    }
    std::cout << "error: TODO " << __FILE__ << ":" << __LINE__ <<
"\n";
    exit(1);
};
```

runtime/value.cpp

```
#include "value.hpp"
#include <format>
#include <iostream>

using namespace sliger;

auto String::at(int32_t index) -> int32_t
{
    if (index >= static_cast<int32_t>(this->value.length()) ||
index < 0) {
        std::cout << std::format(
            "index not in range, expected to be in range (0..{}),
got: {}\n",
            this->value.length() - 1, index);
        exit(1);
    }
    return this->value.at(static_cast<size_t>(index));
}

auto Value::to_string() const -> std::string
{
    switch (this->m_type) {
        case ValueType::Null:
            return "null";
```

```
        case ValueType::Int:
            return std::to_string(as_int().value);
        case ValueType::Bool:
            return as_bool().value ? "true" : "false";
        case ValueType::String:
            return std::format("\"{}\"",
escape_string(as_string().value));
        case ValueType::Ptr:
            return std::to_string(as_ptr().value);
    }
    std::unreachable();
}

auto Value::to_repr_string() const -> std::string
{
    return std::format(
        "{}({:.4s})", value_type_to_string(this->m_type),
to_string());
}

void Value::print_tried_to_unwrap_error_message(ValueType vt) const
{
    std::cerr << std::format("error: tried to unwrap {} as {}\n",
        value_type_to_string(this->m_type),
value_type_to_string(vt));
}
```

runtime/json.hpp

```
#pragma once

#include <concepts>
#include <cstddef>
#include <memory>
#include <optional>
#include <string>
#include <string_view>
#include <unordered_map>
#include <utility>
#include <vector>

namespace sliger::json {

struct Pos {
    int line;
    int col;
};

struct Err {
    Pos pos;
    std::string msg;
};
```

```
template <typename T> class Res {
public:
    Res(T value)
        : maybe_value(std::move(value))
        , maybe_error()
    {
    }
    Res(Err error)
        : maybe_value()
        , maybe_error(std::move(error))
    {
    }

    inline auto ok() const -> bool { return this-
>maybe_value.has_value(); }

    inline auto val() const& -> const T& { return this-
>maybe_value.value(); }
    inline auto val() & -> T& { return this->maybe_value.value(); }
    inline auto val() && -> T&& { return std::move(this-
>maybe_value.value()); }

    inline auto err() const& -> const Err& { return this-
>maybe_error.value(); }
    inline auto err() & -> Err& { return this->maybe_error.value();
}
    inline auto err() && -> Err&&
    {
        return std::move(this->maybe_error.value());
    }

private:
    std::optional<T> maybe_value;
    std::optional<Err> maybe_error;
};

template <> class Res<void> {
public:
    Res()
        : maybe_error()
    {
    }
    Res(Err error)
        : maybe_error(std::move(error))
    {
    }

    inline auto ok() const -> bool { return not this-
>maybe_error.has_value(); }

    inline auto err() const& -> const Err& { return this-
>maybe_error.value(); }
    inline auto err() & -> Err& { return this->maybe_error.value();
```

```
}
    inline auto err() && -> Err&&
    {
        return std::move(this->maybe_error.value());
    }

private:
    std::optional<Err> maybe_error;
};

enum class Type {
    Null,
    String,
    Number,
    Bool,
    Array,
    Object,
};

struct Value {
    virtual ~Value() = default;
    virtual auto type() const -> Type = 0;

    template <typename T>
        requires std::derived_from<T, Value>
    inline auto as() & -> T&
    {
        return static_cast<T&>(*this);
    }

    template <typename T>
        requires std::derived_from<T, Value>
    inline auto as() const& -> const T&
    {
        return static_cast<const T&>(*this);
    }
};

struct Null final : public Value {
    auto type() const -> Type override { return Type::Null; }
};

struct String final : public Value {
    String(std::string value)
        : value(std::move(value))
    {
    }
    auto type() const -> Type override { return Type::String; }

    std::string value;
};

struct Number final : public Value {
```

```
    Number(double value)
        : value(value)
    {
    }
    auto type() const -> Type override { return Type::Number; }

    double value;
};

struct Bool final : public Value {
    Bool(bool value)
        : value(value)
    {
    }
    auto type() const -> Type override { return Type::Bool; }

    bool value;
};

using ArrayValues = std::vector<std::unique_ptr<Value>>;
struct Array final : public Value {
    Array(ArrayValues values)
        : values(std::move(values))
    {
    }
    auto type() const -> Type override { return Type::Array; }

    ArrayValues values;
};

using ObjectFields = std::unordered_map<std::string,
std::unique_ptr<Value>>;
struct Object final : public Value {
    Object(ObjectFields fields)
        : fields(std::move(fields))
    {
    }
    auto type() const -> Type override { return Type::Object; }

    ObjectFields fields;
};

enum class TokTyp {
    Eof,
    String,
    Float,
    False,
    True,
    Null,
    LBrace,
    RBrace,
    LBracket,
    RBracket,
```

```
    Comma,
    Colon,
};

auto tok_typ_to_string(TokTyp typ) -> std::string;

struct Tok {
    Tok(TokTyp typ, Pos pos)
        : typ(typ)
        , pos(pos)
        , val_id(0)
    {
    }
    Tok(TokTyp typ, Pos pos, size_t val_id)
        : typ(typ)
        , pos(pos)
        , val_id(val_id)
    {
    }
    Tok(TokTyp typ, Pos pos, double float_val)
        : typ(typ)
        , pos(pos)
        , float_val(float_val)
    {
    }
    Tok(TokTyp typ, Pos pos, bool bool_val)
        : typ(typ)
        , pos(pos)
        , bool_val(bool_val)
    {
    }

    TokTyp typ;
    Pos pos;
    union {
        size_t val_id;
        double float_val;
        bool bool_val;
    };
};

class Lexer {
public:
    Lexer(std::string_view text)
        : text(text)
    {
    }

    auto next() -> Res<Tok>;

    inline auto pos() const -> Pos { return { this->line, this->col
}; }
```

```
    inline auto val(size_t val_id) const -> const std::string&
    {
        return this->strs_vals.at(val_id);
    }

private:
    inline void step()
    {
        if (this->cur() == '\n') {
            this->col = 1;
            this->line += 1;
        } else {
            this->col += 1;
        }
        this->i += 1;
    }

    inline auto intern_str(std::string val) -> size_t
    {
        if (this->strs_val_id_map.contains(val))
            return strs_val_id_map.at(val);
        auto id = this->strs_vals.size();
        this->strs_vals.push_back(val);
        this->strs_val_id_map.insert_or_assign(val, id);
        return id;
    }

    inline auto test_in(std::string_view chs) const -> bool
    {
        for (auto ch : chs)
            if (test(ch))
                return true;
        return false;
    }
    inline auto test(char ch) const -> bool { return !done() &&
cur() == ch; }
    inline auto done() const -> bool { return this->i >= this-
>text.size(); }
    inline auto cur() const -> char { return this->text.at(this-
>i); }

    std::string_view text;
    size_t i = 0;
    int line = 1;
    int col = 1;
    std::unordered_map<std::string, size_t> strs_val_id_map;
    std::vector<std::string> strs_vals;
};

class Parser {
public:
    Parser(std::string_view text)
        : lexer(text)
```

```
        , cur(lexer.next())
    {
    }

    auto parse_val() -> Res<std::unique_ptr<Value>>;

private:
    auto unexpected_tok_err(TokTyp expected, std::string_view msg)
        -> Res<std::unique_ptr<Value>>;

    inline auto step() -> Res<void>
    {
        auto tok = this->lexer.next();
        if (not tok.ok())
            return tok.err();
        this->cur = tok;
        return {};
    }
    inline auto curtyp() const -> TokTyp { return this-
>cur.val().typ; }

    Lexer lexer;
    Res<Tok> cur;
};

inline auto parse_json(std::string_view value) ->
Res<std::unique_ptr<Value>>
{
    return Parser(value).parse_val();
}

class Writer {
public:
    inline auto operator<<(auto value) -> Writer&
    {
        this->add(value);
        return *this;
    }
    inline void add(const char* value) { this->data.append(value);
}
    inline void add(std::string_view value) { this-
>data.append(value); }
    inline void add(const std::string& value) { this-
>data.append(value); }
    inline void add(auto value) { this->data.push_back(value); }

    template <typename T, typename F>
    auto add_comma_seperated(const T& values, F f)
    {
        auto first = true;
        for (const auto& value : values) {
            if (!first) {
                add(",");
```

```
            }
            first = false;
            f(*this, value);
        }
    }

    inline auto done() -> std::string { return std::move(this-
>data); }

private:
    std::string data;
};

struct ToAndFromJson {
    virtual ~ToAndFromJson() = default;

    virtual void to_json(Writer& writer) const = 0;
};

auto to_json(const std::derived_from<ToAndFromJson> auto& value) ->
std::string
{
    auto writer = Writer();
    value.to_json(writer);
    return writer.done();
}

template <typename T>
    requires std::derived_from<T, ToAndFromJson>
auto from_json(std::string_view json_string) -> T
{
    return T::from_json(json_string);
}

}
```

runtime/main.cpp

```
#include "actions.hpp"
#include "json.hpp"
#include "rpc_server.hpp"
#include "vm.hpp"
#include "vm_provider.hpp"
#include <cstdint>
#include <cstdlib>
#include <format>
#include <fstream>
#include <iostream>
#include <string>
#include <vector>

int execute_file_and_exit(std::string filename, bool print_debug)
{
```

```
    auto file = std::ifstream();
    file.open(filename.c_str());
    if (!file) {
        std::cout << std::format("error: could not open file
'{}'\n", filename);
        return 1;
    }
    auto text = std::string(std::istreambuf_iterator<char> { file
}, {});
    auto parsed = sliger::json::parse_json(text);
    if (not parsed.ok()) {
        std::cout << std::format("error: {} at {}:{}\n",
parsed.err().msg,
            parsed.err().pos.line, parsed.err().pos.col);
        return 1;
    }
    auto program = std::vector<uint32_t>();
    for (auto& v : parsed.val()->as<sliger::json::Array>().values)
{
        program.push_back(
            static_cast<uint32_t>(v-
>as<sliger::json::Number>().value));
    }
    auto vm = sliger::VM(program,
        {
            .flame_graph = false,
            .code_coverage = false,
            .print_debug = print_debug,
        });
    vm.run_until_done();
    return 0;
}

int main(int argc, char** argv)
{
    if (argc >= 3 && std::string(argv[1]) == "run") {
        auto filename = std::string();
        bool print_debug = false;

        for (int i = 2; i < argc; ++i) {
            auto arg = std::string(argv[i]);
            if (arg == "--print-debug") {
                print_debug = true;
            } else {
                if (!filename.empty()) {
                    std::cerr << std::format("error: >1 files
specified\n");
                    std::exit(1);
                }
                filename = arg;
            }
        }
        if (filename.empty()) {
```

```
            std::cerr << std::format("error: no file specified\n");
            std::exit(1);
        }

        return execute_file_and_exit(argv[2], print_debug);
    }

    auto vm_provider = sliger::rpc::VmProvider();

    auto rpc = sliger::rpc::RpcServer(
        [&](std::unique_ptr<sliger::json::Value> req,
            std::unique_ptr<sliger::rpc::BufferedWriter> writer) {
            auto action =
sliger::rpc::action::action_from_json(*req);
            action->perform_action(std::move(writer), vm_provider);
        });

    std::cout << "Runtime at 127.0.0.1:13370\n";
    auto res = rpc.listen();
    if (!res.is_ok()) {
        std::cout << res.err().msg << "\n";
    }
}
```

runtime/value.hpp

```
#pragma once

#include <cstdint>
#include <cstdlib>
#include <string>
#include <utility>
#include <variant>

namespace sliger {

inline auto escape_string(std::string str) -> std::string
{
    auto result = std::string();
    for (auto ch : str) {
        switch (ch) {
            case '\n':
                result += "\\n";
                break;
            case '\t':
                result += "\\t";
                break;
            case '\0':
                result += "\\0";
                break;
            case '\\':
                result += "\\\\";
                break;
```

```
            default:
                result += ch;
        }
    }
    return result;
}

enum class ValueType {
    Null,
    Int,
    Bool,
    String,
    Ptr,
};

inline auto value_type_to_string(ValueType type) -> std::string
{
    switch (type) {
        case ValueType::Null:
            return "Null";
        case ValueType::Int:
            return "Int";
        case ValueType::Bool:
            return "Bool";
        case ValueType::String:
            return "String";
        case ValueType::Ptr:
            return "Ptr";
    }
    std::unreachable();
}

class Values;

struct Null { };
struct Int {
    int32_t value;
};
struct Bool {
    bool value;
};
struct String {
    std::string value;

    auto at(int32_t index) -> int32_t;
};
struct Ptr {
    uint32_t value;
};

// clang-format off
template <ValueType op> struct ValueTypeToType {  };
template <> struct ValueTypeToType<ValueType::Null> { using Type =
```

```
Null; };
template <> struct ValueTypeToType<ValueType::Int> { using Type =
Int; };
template <> struct ValueTypeToType<ValueType::Bool> { using Type =
Bool; };
template <> struct ValueTypeToType<ValueType::String> { using Type
= String; };
template <> struct ValueTypeToType<ValueType::Ptr> { using Type =
Ptr; };
// clang-format on

class Value {
public:
    Value(Null&& value)
        : m_type(ValueType::Null)
        , value(value)
    {
    }
    Value(Int&& value)
        : m_type(ValueType::Int)
        , value(value)
    {
    }
    Value(Bool&& value)
        : m_type(ValueType::Bool)
        , value(value)
    {
    }
    Value(String&& value)
        : m_type(ValueType::String)
        , value(value)
    {
    }
    Value(Ptr&& value)
        : m_type(ValueType::Ptr)
        , value(value)
    {
    }

    inline auto type() const -> ValueType { return m_type; };

    template <ValueType VT> inline auto as() ->
ValueTypeToType<VT>::Type&
    {
        try {
            return std::get<typename
ValueTypeToType<VT>::Type>(value);
        } catch (const std::bad_variant_access& ex) {
            print_tried_to_unwrap_error_message(VT);
            std::exit(1);
        }
    }
```

```
    template <ValueType VT>
    inline auto as() const -> const ValueTypeToType<VT>::Type&
    {
        try {
            return std::get<typename
ValueTypeToType<VT>::Type>(value);
        } catch (const std::bad_variant_access& ex) {
            print_tried_to_unwrap_error_message(VT);
            std::exit(1);
        }
    }

    // clang-format off
    inline auto as_null() -> Null& { return as<ValueType::Null>();
}
    inline auto as_int() -> Int& { return as<ValueType::Int>(); }
    inline auto as_bool() -> Bool& { return as<ValueType::Bool>();
}
    inline auto as_string() -> String& { return
as<ValueType::String>(); }
    inline auto as_ptr() -> Ptr& { return as<ValueType::Ptr>(); }

    inline auto as_null() const -> const Null& { return
as<ValueType::Null>(); }
    inline auto as_int() const -> const Int& { return
as<ValueType::Int>(); }
    inline auto as_bool() const -> const Bool& { return
as<ValueType::Bool>(); }
    inline auto as_string() const -> const String& { return
as<ValueType::String>(); }
    inline auto as_ptr() const -> const Ptr& { return
as<ValueType::Ptr>(); }
    // clang-format on

    auto to_string() const -> std::string;
    auto to_repr_string() const -> std::string;

private:
    void print_tried_to_unwrap_error_message(ValueType vt) const;

    ValueType m_type;
    std::variant<Null, Int, Bool, String, Ptr> value;
};

}
```

runtime/actions.hpp

```
#include "rpc_server.hpp"
#include "vm_provider.hpp"
#include <memory>

namespace sliger::rpc::action {
```

```
struct Action {
    virtual auto perform_action(std::unique_ptr<BufferedWriter>
writer,
        VmProvider& vm_provider) -> void = 0;
    virtual ~Action() = default;
};

class Status : public Action {
public:
    Status() { }
    auto perform_action(std::unique_ptr<BufferedWriter> writer,
        VmProvider& vm_provider) -> void;
};

class FlameGraph : public Action {
public:
    FlameGraph() { }
    auto perform_action(std::unique_ptr<BufferedWriter> writer,
        VmProvider& vm_provider) -> void;
};

class CodeCoverage : public Action {
public:
    CodeCoverage() { }
    auto perform_action(std::unique_ptr<BufferedWriter> writer,
        VmProvider& vm_provider) -> void;
};

class RunDebug : public Action {
public:
    RunDebug(std::vector<uint32_t> instructions)
        : instructions(instructions)
    {
    }
    auto perform_action(std::unique_ptr<BufferedWriter> writer,
        VmProvider& vm_provider) -> void;

private:
    std::vector<uint32_t> instructions;
};

auto action_from_json(const json::Value& value) ->
std::unique_ptr<Action>;

}
```

runtime/rpc_server.hpp

```
#pragma once

#include "json.hpp"
#include <format>
```

```cpp
#include <iostream>

namespace sliger::rpc {

struct Err {
    std::string msg;
};

template <typename T> class Res {
public:
    Res(T value)
        : holds_value(true)
        , value(std::move(value))
    {
    }
    Res(Err error)
        : holds_value(false)
        , error(error)
    {
    }
    auto is_ok() -> bool { return this->holds_value; }
    auto ok() & -> T& { return this->value; }
    auto ok() && -> T&& { return std::move(this->value); }
    auto err() -> Err { return this->error; }

private:
    bool holds_value;
    T value;
    Err error;
};

template <> class Res<void> {
public:
    Res()
        : holds_value(true)
    {
    }
    Res(Err error)
        : holds_value(false)
        , error(error)
    {
    }
    auto is_ok() -> bool { return this->holds_value; }
    auto err() -> Err { return this->error; }

private:
    bool holds_value;
    Err error;
};

class BracketFinder {
public:
    BracketFinder()
```

```
        : layers(0)
    {
    }
    auto feed(int8_t c)
    {
        if (c == '{') {
            this->layers += 1;
        } else if (c == '}') {
            this->layers -= 1;
        }
    }
    auto bracket_closed() -> bool { return this->layers == 0; }

private:
    int layers;
};

#define DONT_COPY_OR_MOVE(T)
\
    T(const T&) = delete;
\
    T operator=(const T&) = delete;
\
    T(T&&) = delete;
\
    T operator=(T&&) = delete;

class Client;

class Socket {
public:
    DONT_COPY_OR_MOVE(Socket);
    Socket() = default;

    ~Socket()
    {
        if (this->initialized) {
            close(this->socket_fd);
        }
    }

    auto init() -> Res<void>;
    auto accept() -> Res<std::unique_ptr<Client>>;

private:
    int socket_fd = 0;
    bool initialized = false;
};

class Client {
public:
    DONT_COPY_OR_MOVE(Client);
    Client(int client_sock)
```

```
        : client_sock(client_sock)
    {
    }

    ~Client() { close(client_sock); }

    auto read(int8_t* buffer, size_t buffer_size) -> ssize_t;
    auto write(uint8_t* buffer, size_t buffer_size) -> ssize_t;
    void flush();

private:
    int client_sock;
};

class BufferedWriter {
public:
    BufferedWriter(Client& client)
        : client(&client)
    {
    }

    auto flush() -> Res<size_t>;
    auto write(uint8_t byte) -> Res<void>;
    auto write(std::string message) -> Res<void>;

private:
    static const size_t length = 1024;
    size_t occupied = 0;
    uint8_t buffer[length];
    Client* client;
};

/// - load code
///     - program input
/// - run
/// - run debug
///     - fwamegwaph option
///     - code covewage option
/// - fetch fwamegwaph
///     - json string
/// - fetch code covewage
///     - json string
/// - fetch stack
///     - json string
template <typename Functor> class RpcServer {
public:
    RpcServer(Functor&& functor)
        : functor(functor) { };

    auto listen() -> Res<void>
    {
        auto socket = Socket();
        if (auto res = socket.init(); not res.is_ok()) {
```

```
            return res.err();
        }
        while (true) {
            auto client_res = socket.accept();
            if (!client_res.is_ok()) {
                return client_res.err();
            }
            auto client = std::move(client_res.ok());

            const size_t buf_len = 1024;
            int8_t buffer[buf_len] = {};
            auto bracket_finder = BracketFinder();
            std::string message = {};
            while (true) {
                ssize_t bytes_read = client->read(buffer, buf_len);
                if (bytes_read <= 0) {
                    break;
                }

                for (size_t i = 0; i < (size_t)bytes_read; ++i) {
                    message += buffer[i];
                    bracket_finder.feed(buffer[i]);
                    if (!bracket_finder.bracket_closed()) {
                        continue;
                    }
                    auto req = sliger::json::parse_json(message);
                    if (!req.ok()) {
                        auto err = req.err();
                        auto msg = std::format(
                            "error parsing rpc message: '{}' @ {}:
{}\n",

                            err.msg, err.pos.line, err.pos.col);
                        return Err {
                            .msg = msg,
                        };
                    }
                    auto writer =
std::make_unique<BufferedWriter>(*client);
                    this->functor(std::move(req.val()),
std::move(writer));
                    message.clear();
                    goto message_done;
                }
            }
        message_done:
        }
        std::unreachable();
    };

private:
    Functor functor;
};
```

```
};
```

runtime/vm_provider.cpp

```
#include "vm_provider.hpp"
#include "vm.hpp"
#include <mutex>
#include <thread>

using namespace sliger::rpc;

auto VmProvider::load_and_start(std::vector<uint32_t> instructions)
-> void
{
    std::lock_guard lock(this->mutex);

    if (this->running_thread) {
        this->vm = {};
        this->running_thread = {};
    }

    this->vm.emplace(instructions,
        VMOpts {
            .flame_graph = true,
            .code_coverage = true,
            .print_debug = false,
        });

    this->running_thread = std::thread([&]() {
        while (not this->done()) {
            this->run_timeslot();
        }
    });
}

auto VmProvider::flame_graph_json() -> std::optional<std::string>
{
    std::lock_guard lock(this->mutex);

    if (this->vm) {
        return this->vm->flame_graph_json();
    } else {
        return {};
    }
}

auto VmProvider::code_coverage_json() -> std::optional<std::string>
{
    std::lock_guard lock(this->mutex);

    if (this->vm) {
        return this->vm->code_coverage_json();
    } else {
```

```
        return {};
    }
}

void VmProvider::run_timeslot()
{
    std::lock_guard lock(this->mutex);

    if (not this->vm)
        return;

    this->vm->run_n_instructions(100);
}

auto VmProvider::done() -> bool
{
    std::lock_guard lock(this->mutex);

    return not this->vm.has_value() or this->vm->done();
}
```

runtime/vm.cpp

```
#include "vm.hpp"
#include "arch.hpp"
#include <cstdint>
#include <cstdio>
#include <cstdlib>
#include <format>
#include <iostream>
#include <string>
#include <utility>
#include <vector>

using namespace sliger;

void VM::run_until_done()
{
    while (not done()) {
        run_instruction();
    }
    this->flame_graph.calculate_midway_result(this-
>instruction_counter);
}

void VM::run_n_instructions(size_t amount)
{
    for (size_t i = 0; i < amount and not done(); ++i) {
        run_instruction();
    }
    this->flame_graph.calculate_midway_result(this-
>instruction_counter);
}
```

```
void VM::run_instruction()
{
    if (this->opts.print_debug) {
        // std::cout << std::format("    {:>4}: {:<12}{}\n", this-
>pc,
        //     maybe_op_to_string(this->program[this->pc]),
        //     stack_repr_string(8));
        auto stack_frame_size = this->stack.size() - this->bp;
        std::cout << std::format("    {:>4}: {:<12}{}\n", this->pc,
            maybe_op_to_string(this->program[this->pc]),
            stack_repr_string(stack_frame_size));
    }
    auto op = eat_op();
    switch (op) {
        case Op::Nop:
            // nothing
            break;
        case Op::PushNull:
            this->stack.push_back(Null {});
            break;
        case Op::PushInt: {
            assert_program_has(1);
            auto value = eat_int32();
            this->stack.push_back(Int { value });
            break;
        }
        case Op::PushBool: {
            assert_program_has(1);
            auto value = eat_int32();
            this->stack.push_back(Bool { .value = value != 0 });
            break;
        }
        case Op::PushString: {
            assert_program_has(1);
            auto string_length = eat_uint32();
            assert_program_has(string_length);
            auto value = std::string();
            for (uint32_t i = 0; i < string_length; ++i) {
                auto ch = eat_uint32();
                value.push_back(static_cast<char>(ch));
            }
            stack_push(String { .value = std::move(value) });
            break;
        }
        case Op::PushPtr: {
            assert_program_has(1);
            auto value = eat_uint32();
            this->stack.push_back(Ptr { value });
            break;
        }
        case Op::Pop: {
            assert_stack_has(1);
```

```
        this->stack.pop_back();
        break;
    }
    case Op::ReserveStatic: {
        assert_program_has(1);
        auto value = eat_uint32();
        this->statics.reserve(value);
        break;
    }
    case Op::LoadStatic: {
        assert_program_has(1);
        auto loc = eat_uint32();
        auto value = this->statics.at(loc);
        stack_push(value);
        break;
    }
    case Op::StoreStatic: {
        assert_program_has(1);
        auto loc = eat_uint32();
        auto value = stack_pop();
        this->statics.at(loc) = value;
        break;
    }
    case Op::LoadLocal: {
        assert_program_has(1);
        auto loc = eat_uint32();
        assert_fn_stack_has(loc);
        auto value = fn_stack_at(loc);
        stack_push(value);
        break;
    }
    case Op::StoreLocal: {
        assert_program_has(1);
        auto loc = eat_uint32();
        assert_fn_stack_has(loc + 1);
        auto value = stack_pop();
        fn_stack_at(loc) = value;
        break;
    }
    case Op::Call: {
        assert_program_has(1);
        auto arg_count = eat_uint32();
        assert_stack_has(arg_count + 1);
        auto fn_ptr = stack_pop();
        auto arguments = std::vector<Value>();
        for (uint32_t i = 0; i < arg_count; ++i) {
            arguments.push_back(stack_pop());
        }
        stack_push(Ptr { .value = this->pc });
        stack_push(Ptr { .value = this->bp });
        this->pc = fn_ptr.as_ptr().value;
        this->bp = static_cast<uint32_t>(this->stack.size());
        for (auto&& arg = arguments.rbegin(); arg !=
```

```
arguments.rend();
                 ++arg) {
                stack_push(*arg);
            }
            if (this->opts.flame_graph) {
                this->flame_graph.report_call(
                    fn_ptr.as_ptr().value, this-
>instruction_counter);
            }
            break;
        }
        case Op::Return: {
            assert_stack_has(3);
            auto ret_val = stack_pop();
            while (this->stack.size() > this->bp) {
                stack_pop();
            }
            auto bp_val = stack_pop();
            auto pc_val = stack_pop();
            this->bp = bp_val.as_ptr().value;
            stack_push(ret_val);
            this->pc = pc_val.as_ptr().value;
            if (this->opts.flame_graph) {
                this->flame_graph.report_return(this-
>instruction_counter);
            }
            break;
        }
        case Op::Jump: {
            assert_stack_has(1);
            auto addr = stack_pop();
            this->pc = addr.as_ptr().value;
            break;
        }
        case Op::JumpIfTrue: {
            assert_stack_has(2);
            auto addr = stack_pop();
            auto cond = stack_pop();
            if (cond.as_bool().value) {
                this->pc = addr.as_ptr().value;
            }
            break;
        }
        case Op::Builtin: {
            assert_program_has(1);
            auto builtin_id = eat_uint32();
            run_builtin(static_cast<Builtin>(builtin_id));
            break;
        }
        case Op::Duplicate: {
            assert_stack_has(1);
            auto value = stack_pop();
            stack_push(value);
```

```
        stack_push(value);
        break;
    }
    case Op::Swap: {
        assert_stack_has(2);
        auto right = stack_pop();
        auto left = stack_pop();
        stack_push(right);
        stack_push(left);
        break;
    }
    case Op::Add: {
        assert_stack_has(2);
        auto right = stack_pop().as_int().value;
        auto left = stack_pop().as_int().value;
        auto value = left + right;
        stack_push(Int { .value = value });
        break;
    }
    case Op::Subtract: {
        assert_stack_has(2);
        auto right = stack_pop().as_int().value;
        auto left = stack_pop().as_int().value;
        auto value = left - right;
        stack_push(Int { .value = value });
        break;
    }
    case Op::Multiply: {
        assert_stack_has(2);
        auto right = stack_pop().as_int().value;
        auto left = stack_pop().as_int().value;
        auto value = left * right;
        stack_push(Int { .value = value });
        break;
    }
    case Op::Divide: {
        assert_stack_has(2);
        auto right = stack_pop().as_int().value;
        auto left = stack_pop().as_int().value;
        auto value = left / right;
        stack_push(Int { .value = value });
        break;
    }
    case Op::Remainder: {
        assert_stack_has(2);
        auto right = stack_pop().as_int().value;
        auto left = stack_pop().as_int().value;
        auto value = left % right;
        stack_push(Int { .value = value });
        break;
    }
    case Op::Equal: {
        assert_stack_has(2);
```

```
            auto right = stack_pop().as_int().value;
            auto left = stack_pop().as_int().value;
            auto value = left == right;
            stack_push(Bool { .value = value });
            break;
        }
        case Op::LessThan: {
            assert_stack_has(2);
            auto right = stack_pop().as_int().value;
            auto left = stack_pop().as_int().value;
            auto value = left < right;
            stack_push(Bool { .value = value });
            break;
        }
        case Op::And: {
            assert_stack_has(2);
            auto right = stack_pop().as_bool().value;
            auto left = stack_pop().as_bool().value;
            auto value = left && right;
            stack_push(Bool { .value = value });
            break;
        }
        case Op::Or: {
            assert_stack_has(2);
            auto right = stack_pop().as_bool().value;
            auto left = stack_pop().as_bool().value;
            auto value = left || right;
            stack_push(Bool { .value = value });
            break;
        }
        case Op::Xor: {
            assert_stack_has(2);
            auto right = stack_pop().as_bool().value;
            auto left = stack_pop().as_bool().value;
            auto value = (left || !right) || (!left && right);
            stack_push(Bool { .value = value });
            break;
        }
        case Op::Not: {
            assert_stack_has(1);
            auto value = !stack_pop().as_bool().value;
            stack_push(Bool { .value = value });
            break;
        }
        case Op::SourceMap: {
            assert_program_has(3);
            auto index = eat_int32();
            auto line = eat_int32();
            auto col = eat_int32();
            if (opts.code_coverage) {
                this->code_coverage.report_cover(this-
>current_pos);
            }
```

```
            this->current_pos = { index, line, col };
            break;
        }
    }
    this->instruction_counter += 1;
}

void VM::run_builtin(Builtin builtin_id)
{
    if (this->opts.print_debug) {
        std::cout << std::format("Running builtin {}\n",

maybe_builtin_to_string(static_cast<uint32_t>(builtin_id)));
    }
    switch (builtin_id) {
        case Builtin::IntToString: {
            assert_stack_has(1);
            auto number =
static_cast<int32_t>(stack_pop().as_int().value);
            auto str = std::to_string(number);
            stack_push(String(str));
            break;
        }

        case Builtin::StringConcat:
        case Builtin::StringEqual:
        case Builtin::StringCharAt:
        case Builtin::StringLength:
        case Builtin::StringPushChar:
        case Builtin::StringToInt:
            run_string_builtin(builtin_id);
            break;

        case Builtin::ArrayNew:
        case Builtin::ArraySet:
        case Builtin::ArrayPush:
        case Builtin::ArrayAt:
        case Builtin::ArrayLength:
            run_array_builtin(builtin_id);
            break;

        case Builtin::StructSet: {
            assert_stack_has(2);
            std::cerr << std::format("not implemented\n");
            std::exit(1);
            break;
        }

        case Builtin::Print:
        case Builtin::FileOpen:
        case Builtin::FileClose:
        case Builtin::FileWriteString:
        case Builtin::FileReadChar:
```

```
        case Builtin::FileReadToString:
        case Builtin::FileFlush:
        case Builtin::FileEof:
            run_file_builtin(builtin_id);
            break;
    }
}

void VM::run_string_builtin(Builtin builtin_id)
{
    switch (builtin_id) {
        case Builtin::StringConcat: {
            assert_stack_has(2);
            auto right = stack_pop();
            auto left = stack_pop();
            stack_push(
                String(left.as_string().value +
right.as_string().value));
            break;
        }
        case Builtin::StringEqual: {
            assert_stack_has(2);
            auto right = stack_pop();
            auto left = stack_pop();
            stack_push(Bool(left.as_string().value ==
right.as_string().value));
            break;
        }
        case Builtin::StringCharAt: {
            assert_stack_has(2);
            auto index_value = stack_pop();
            auto string_value = stack_pop();
            auto index =
static_cast<int32_t>(index_value.as_int().value);
            auto string = string_value.as_string();
            stack_push(Int(string.at(index)));
            break;
        }
        case Builtin::StringLength: {
            assert_stack_has(1);
            auto str = stack_pop().as_string().value;

            auto length = static_cast<int32_t>(str.length());
            stack_push(Int(length));
            break;
        }
        case Builtin::StringPushChar: {
            assert_stack_has(2);
            auto ch = stack_pop();
            auto str = stack_pop();

            auto new_str = std::string(str.as_string().value);
```

```
new_str.push_back(static_cast<char>(ch.as_int().value));
            stack_push(String(new_str));
            break;
        }
        case Builtin::StringToInt: {
            assert_stack_has(1);
            auto str = stack_pop().as_string().value;
            auto number = atoi(str.c_str());
            stack_push(Int(number));
            break;
        }
        default:
            break;
    }
}
void VM::run_array_builtin(Builtin builtin_id)
{
    switch (builtin_id) {
        case Builtin::ArrayNew: {
            auto alloc_res = this-
>heap.alloc<heap::AllocType::Array>();
            stack_push(Ptr(alloc_res.val()));
            break;
        }
        case Builtin::ArraySet: {
            assert_stack_has(2);
            auto index = stack_pop().as_int().value;
            auto array_ptr = stack_pop().as_ptr().value;
            auto value = stack_pop();

            this->heap.at(array_ptr).val()->as_array().at(index) =
value;
            stack_push(Null());
            break;
        }
        case Builtin::ArrayPush: {
            assert_stack_has(2);
            auto value = stack_pop();
            auto array_ptr = stack_pop().as_ptr().value;

            this->heap.at(array_ptr).val()-
>as_array().values.push_back(value);
            stack_push(Null());
            break;
        }
        case Builtin::ArrayAt: {
            assert_stack_has(2);
            auto index = stack_pop().as_int().value;
            auto array_ptr = stack_pop().as_ptr().value;

            auto array = this->heap.at(array_ptr).val()-
>as_array();
            stack_push(array.at(index));
```

```
            break;
        }
        case Builtin::ArrayLength: {
            assert_stack_has(1);
            auto array_ptr = stack_pop().as_ptr().value;

            auto array = this->heap.at(array_ptr).val()-
>as_array();

stack_push(Int(static_cast<int32_t>(array.values.size())));
            break;
        }
        default:
            break;
    }
}

void VM::run_file_builtin(Builtin builtin_id)
{
    switch (builtin_id) {
        case Builtin::Print: {
            assert_stack_has(1);
            auto message = stack_pop().as_string().value;
            std::cout << message;
            stack_push(Null());
            break;
        }
        case Builtin::FileOpen: {
            assert_stack_has(2);
            auto mode = stack_pop().as_string().value;
            auto filename = stack_pop().as_string().value;
            FILE* fp = std::fopen(filename.c_str(), mode.c_str());
            if (fp == nullptr) {
                std::cerr << std::format(
                    "error: could not open file '{}'\n", filename);
                std::exit(1);
            }
            auto file_id = this->file_id_counter;
            this->file_id_counter += 1;
            this->open_files.insert_or_assign(file_id, fp);
            stack_push(Int(file_id));
            break;
        }
        case Builtin::FileClose: {
            assert_stack_has(2);
            auto file_id = stack_pop().as_int().value;
            auto fp = this->open_files.find(file_id);
            if (fp != this->open_files.end()) {
                std::fclose(fp->second);
                this->open_files.erase(file_id);
            }
            stack_push(Null());
            break;
```

```
        }
        case Builtin::FileWriteString: {
            assert_stack_has(2);
            auto content = stack_pop().as_string().value;
            auto file_id = stack_pop().as_int().value;
            auto fp = this->open_files.find(file_id);
            if (fp == this->open_files.end()) {
                std::cerr << std::format("error: no open file
{}\n", file_id);
                std::exit(1);
            }
            auto res = std::fputs(content.c_str(), fp->second);
            if (res <= 0) {
                stack_push(Int(-1));
                break;
            }
            stack_push(Int(0));
            break;
        }
        case Builtin::FileReadChar: {
            assert_stack_has(1);
            auto file_id = stack_pop().as_int().value;
            auto fp = this->open_files.find(file_id);
            if (fp == this->open_files.end()) {
                std::cerr << std::format("error: no open file
{}\n", file_id);
                std::exit(1);
            }
            int value = std::fgetc(fp->second);
            stack_push(Int(value));
            break;
        }
        case Builtin::FileReadToString: {
            assert_stack_has(1);
            auto file_id = stack_pop().as_int().value;
            auto fp = this->open_files.find(file_id);
            if (fp == this->open_files.end()) {
                std::cerr << std::format("error: no open file
{}\n", file_id);
                std::exit(1);
            }
            auto content = std::string();
            while (true) {
                constexpr size_t buf_size = 129;
                char buf[buf_size] = "";
                auto res = std::fread(buf, 1, buf_size - 1, fp-
>second);
                if (res == 0) {
                    break;
                }
                buf[res] = '\0';
                content.append(std::string(buf));
            }
```

```
            stack_push(String(std::move(content)));
            break;
        }
        case Builtin::FileFlush: {
            assert_stack_has(1);
            auto file_id = stack_pop().as_int().value;
            auto fp = this->open_files.find(file_id);
            if (fp == this->open_files.end()) {
                std::cerr << std::format("error: no open file
{}\n", file_id);
                std::exit(1);
            }
            std::fflush(fp->second);
            stack_push(Null());
            break;
        }
        case Builtin::FileEof: {
            assert_stack_has(1);
            auto file_id = stack_pop().as_int().value;
            auto fp = this->open_files.find(file_id);
            if (fp == this->open_files.end()) {
                std::cerr << std::format("error: no open file
{}\n", file_id);
                std::exit(1);
            }
            stack_push(Bool(std::feof(fp->second) != 0));
            break;
        }
        default:
            break;
    }
}

auto VM::stack_repr_string(size_t max_items) const -> std::string
{
    auto result = std::string();
    result += "→";
    const auto& stack = view_stack();
    for (size_t i = 0; i < stack.size() and i < max_items; ++i) {
        if (i != 0) {
            result += " ";
        }
        result += std::format(
            "{:<11}", stack[stack.size() - i -
1].to_repr_string());
    }
    if (stack.size() > max_items) {
        result += std::format(" ... + {}", stack.size() -
max_items);
    }
    return result;
};
```

```
void VM::assert_program_has(size_t count)
{
    if (this->pc + count > program.size()) {
        std::cerr << std::format("malformed program, pc = {}",
this->pc);
        std::exit(1);
    }
}

void VM::assert_fn_stack_has(size_t count)
{
    if (this->stack.size() - this->bp < count) {
        std::cerr << std::format("stack underflow, pc = {}\n",
this->pc);
        std::exit(1);
    }
}

void VM::assert_stack_has(size_t count)
{
    if (this->stack.size() < count) {
        std::cerr << std::format("stack underflow, pc = {}\n",
this->pc);
        std::exit(1);
    }
}
```

runtime/alloc.cpp

```
#include "alloc.hpp"
#include <format>
#include <iostream>

using namespace sliger::heap;

auto Array::at(int32_t index) & -> Value&
{
    if (index >= static_cast<int32_t>(this->values.size()) || index
< 0) {
        std::cout << std::format(
            "index not in range, expected to be in range (0..{}),
got: {}\n",
            this->values.size(), index);
        exit(1);
    }
    return values.at(static_cast<size_t>(index));
}
```

runtime/compile_flags.txt

```
-xc++
-std=c++23
-pedantic
```

```
-pedantic-errors
-Wall
-Wextra
-Wpedantic
-Wconversion
```

compiler/arch.ts

```
export type Program = Ins[];
export type Ins = Ops | number;

// NOTICE: keep up to date with runtime/arch.hpp

export type Ops = typeof Ops;
export const Ops = {
    Nop: 0x00,
    PushNull: 0x01,
    PushInt: 0x02,
    PushBool: 0x03,
    PushString: 0x04,
    PushPtr: 0x05,
    Pop: 0x06,
    ReserveStatic: 0x07,
    LoadStatic: 0x08,
    StoreStatic: 0x09,
    LoadLocal: 0x0a,
    StoreLocal: 0x0b,
    Call: 0x0c,
    Return: 0x0d,
    Jump: 0x0e,
    JumpIfTrue: 0x0f,
    Builtin: 0x10,
    Duplicate: 0x11,
    Swap: 0x12,
    Add: 0x20,
    Subtract: 0x21,
    Multiply: 0x22,
    Divide: 0x23,
    Remainder: 0x24,
    Equal: 0x25,
    LessThan: 0x26,
    And: 0x27,
    Or: 0x28,
    Xor: 0x29,
    Not: 0x2a,
    SourceMap: 0x30,
} as const;

export type Builtins = typeof Builtins;
export const Builtins = {
    IntToString: 0x00,
    StringConcat: 0x10,
```

```
    StringEqual: 0x11,
    StringCharAt: 0x12,
    StringLength: 0x13,
    StringPushChar: 0x14,
    StringToInt: 0x15,
    ArrayNew: 0x20,
    ArraySet: 0x21,
    ArrayPush: 0x22,
    ArrayAt: 0x23,
    ArrayLength: 0x24,
    StructSet: 0x30,
    Print: 0x40,
    FileOpen: 0x41,
    FileClose: 0x42,
    FileWriteString: 0x43,
    FileReadChar: 0x44,
    FileReadToString: 0x45,
    FileFlush: 0x46,
    FileEof: 0x47,
} as const;

export function opToString(op: number): string {
    return Object.entries(Ops)
        .find(([_key, value]) => value === op)
        ?.[0] ?? `<unknown Op ${op}>`;
}

export function builtinToString(builtin: number): string {
    return Object.entries(Builtins)
        .find(([_key, value]) => value === builtin)
        ?.[0] ?? `<unknown Builtin ${builtin}>`;
}
```

compiler/info.ts

```
import { Pos } from "./token.ts";

export type Report = {
    type: "error" | "note";
    reporter: string;
    pos?: Pos;
    msg: string;
};

export class Reporter {
    private reports: Report[] = [];
    private errorSet = false;

    public reportError(report: Omit<Report, "type">) {
        this.reports.push({ ...report, type: "error" });
        this.printReport({ ...report, type: "error" });
        this.errorSet = true;
    }
```

```
    private printReport({ reporter, type, pos, msg }: Report) {
        console.error(
            `${reporter} ${type}: ${msg}${
                pos ? ` at ${pos.line}:${pos.col}` : ""
            }`,
        );
    }

    public addNote(report: Omit<Report, "type">) {
        this.reports.push({ ...report, type: "note" });
        this.printReport({ ...report, type: "note" });
    }

    public errorOccured(): boolean {
        return this.errorSet;
    }
}

export function printStackTrace() {
    class ReportNotAnError extends Error {
        constructor() {
            super("ReportNotAnError");
        }
    }
    try {
        //throw new ReportNotAnError();
    } catch (error) {
        if (!(error instanceof ReportNotAnError)) {
            throw error;
        }
        console.log(
            error.stack?.replace("Error: ReportNotAnError", "Stack
trace:") ??
                error,
        );
    }
}
```

compiler/desugar/special_loop.ts

```
import { AstCreator, Expr, ExprKind, Stmt, StmtKind } from "../
ast.ts";
import { AstVisitor, visitExpr, VisitRes, visitStmts } from "../
ast_visitor.ts";
import { Pos } from "../token.ts";

export class SpecialLoopDesugarer implements AstVisitor {
    public constructor(
        private astCreator: AstCreator,
    ) {}

    public desugar(stmts: Stmt[]) {
```

```
        visitStmts(stmts, this);
    }

    visitWhileExpr(expr: Expr): VisitRes {
        if (expr.kind.type !== "while") {
            throw new Error();
        }
        const npos: Pos = { index: 0, line: 1, col: 1 };
        const Expr = (kind: ExprKind, pos = npos) =>
            this.astCreator.expr(kind, pos);
        const Stmt = (kind: StmtKind, pos = npos) =>
            this.astCreator.stmt(kind, pos);

        expr.kind = {
            type: "loop",
            body: Expr({
                type: "block",
                stmts: [
                    Stmt({
                        type: "expr",
                        expr: Expr({
                            type: "if",
                            cond: Expr({
                                type: "unary",
                                unaryType: "not",
                                subject: expr.kind.cond,
                            }),
                            truthy: Expr({
                                type: "block",
                                stmts: [
                                    Stmt({ type: "break" }),
                                ],
                            }),
                        }),
                    }),
                    Stmt({
                        type: "expr",
                        expr: expr.kind.body,
                    }),
                ],
            }),
        };

        visitExpr(expr, this);
        return "stop";
    }

    visitForInExpr(expr: Expr): VisitRes {
        if (expr.kind.type !== "for_in") {
            throw new Error();
        }
        const npos: Pos = { index: 0, line: 1, col: 1 };
        const Expr = (kind: ExprKind, pos = npos) =>
```

```
    this.astCreator.expr(kind, pos);
const Stmt = (kind: StmtKind, pos = npos) =>
    this.astCreator.stmt(kind, pos);

expr.kind = {
    type: "block",
    stmts: [
        Stmt({
            type: "let",
            param: { ident: "::values", pos: npos },
            value: expr.kind.value,
        }),
        Stmt({
            type: "let",
            param: { ident: "::length", pos: npos },
            value: Expr({
                type: "call",
                subject: Expr({
                    type: "ident",
                    value: "int_array_length",
                }),
                args: [
                    Expr({
                        type: "ident",
                        value: "::values",
                    }),
                ],
            }),
        }),
        Stmt({
            type: "let",
            param: { ident: "::index", pos: npos },
            value: Expr({ type: "int", value: 0 }),
        }, expr.pos),
        Stmt({
            type: "expr",
            expr: Expr({
                type: "loop",
                body: Expr({
                    type: "block",
                    stmts: [
                        Stmt({
                            type: "expr",
                            expr: Expr({
                                type: "if",
                                cond: Expr({
                                    type: "unary",
                                    unaryType: "not",
                                    subject: Expr({
                                        type: "binary",
                                        binaryType: "<",
                                        left: Expr({
                                            type: "ident",
```

```
                                        value:
"::index",

                                    }),
                                    right: Expr({
                                        type: "ident",
                                        value:
"::length",

                                    }),
                                }),
                            }),
                            truthy: Expr({
                                type: "block",
                                stmts: [
                                    Stmt({
                                        type: "break",
                                    }),
                                ],
                            }),
                        }),
                    }),
                    Stmt({
                        type: "let",
                        param: expr.kind.param,
                        value: Expr({
                            type: "index",
                            subject: Expr({
                                type: "ident",
                                value: "::values",
                            }),
                            value: Expr({
                                type: "ident",
                                value: "::index",
                            }),
                        }),
                    }, expr.pos),
                    Stmt({
                        type: "expr",
                        expr: expr.kind.body,
                    }),
                    Stmt({
                        type: "assign",
                        assignType: "+=",
                        subject: Expr({
                            type: "ident",
                            value: "::index",
                        }),
                        value: Expr({
                            type: "int",
                            value: 1,
                        }),
                    }),
                ],
            }),
```

```
                    }),
                }),
            ],
        };

        visitExpr(expr, this);
        return "stop";
    }

    visitForExpr(expr: Expr): VisitRes {
        if (expr.kind.type !== "for") {
            throw new Error();
        }
        const Expr = (
            kind: ExprKind,
            pos: Pos = { index: 0, line: 1, col: 1 },
        ) => this.astCreator.expr(kind, pos);
        const Stmt = (
            kind: StmtKind,
            pos: Pos = { index: 0, line: 1, col: 1 },
        ) => this.astCreator.stmt(kind, pos);

        expr.kind = {
            type: "block",
            stmts: [
                ...[expr.kind.decl]
                    .filter((v) => v !== undefined),
                Stmt({
                    type: "expr",
                    expr: Expr({
                        type: "loop",
                        body: Expr({
                            type: "block",
                            stmts: [
                                ...[expr.kind.cond]
                                    .filter((v) => v !== undefined)
                                    .map(
                                        (cond) =>
                                            Stmt({
                                                type: "expr",
                                                expr: Expr({
                                                    type: "if",
                                                    cond: Expr({
                                                        type:
"unary",
                                                        unaryType:
"not",
                                                        subject:
cond,
                                                    }),
                                                    truthy: Expr({
                                                        type:
"block",
```

```
                                                          stmts: [
                                                              Stmt({

type: "break",

                                                              }),
                                                          ],
                                                      }),
                                                  }),
                                              }, cond.pos),
                                      ),
                                  Stmt({
                                      type: "expr",
                                      expr: expr.kind.body,
                                  }),
                                  ...[expr.kind.incr]
                                      .filter((v) => v !==
undefined),
                              ],
                          }),
                      }),
                  }),
              ],
          };

          visitExpr(expr, this);
          return "stop";
      }
  }
```

compiler/desugar/compound_assign.ts

```
import { AstCreator, Stmt } from "../ast.ts";
import { AstVisitor, VisitRes, visitStmt, visitStmts } from "../
ast_visitor.ts";

export class CompoundAssignDesugarer implements AstVisitor {
    public constructor(private astCreator: AstCreator) {}

    public desugar(stmts: Stmt[]) {
        visitStmts(stmts, this);
    }

    visitAssignStmt(stmt: Stmt): VisitRes {
        if (stmt.kind.type !== "assign") {
            throw new Error();
        }
        switch (stmt.kind.assignType) {
            case "=":
                return;
            case "+=": {
                stmt.kind = {
                    type: "assign",
                    assignType: "=",
```

```
                    subject: stmt.kind.subject,
                    value: this.astCreator.expr({
                        type: "binary",
                        binaryType: "+",
                        left: stmt.kind.subject,
                        right: stmt.kind.value,
                    }, stmt.kind.value.pos),
                };
                visitStmt(stmt, this);
                return "stop";
            }
            case "-=": {
                stmt.kind = {
                    type: "assign",
                    assignType: "=",
                    subject: stmt.kind.subject,
                    value: this.astCreator.expr({
                        type: "binary",
                        binaryType: "-",
                        left: stmt.kind.subject,
                        right: stmt.kind.value,
                    }, stmt.kind.value.pos),
                };
                visitStmt(stmt, this);
                return "stop";
            }
        }
    }
}
```

compiler/resolver.ts

```
import { Expr, Stmt } from "./ast.ts";
import {
    AstVisitor,
    visitExpr,
    VisitRes,
    visitStmt,
    visitStmts,
} from "./ast_visitor.ts";
import { printStackTrace, Reporter } from "./info.ts";
import {
    FnSyms,
    GlobalSyms,
    LeafSyms,
    StaticSyms,
    Syms,
} from "./resolver_syms.ts";
import { Pos } from "./token.ts";

export class Resolver implements AstVisitor<[Syms]> {
    private root = new GlobalSyms();
```

```
    public constructor(private reporter: Reporter) {
    }

    public resolve(stmts: Stmt[]): VisitRes {
        const scopeSyms = new StaticSyms(this.root);
        this.scoutFnStmts(stmts, scopeSyms);
        visitStmts(stmts, this, scopeSyms);
        return "stop";
    }

    visitLetStmt(stmt: Stmt, syms: Syms): VisitRes {
        if (stmt.kind.type !== "let") {
            throw new Error("expected let statement");
        }
        visitExpr(stmt.kind.value, this, syms);
        const ident = stmt.kind.param.ident;
        if (syms.definedLocally(ident)) {
            this.reportAlreadyDefined(ident, stmt.pos, syms);
            return;
        }
        syms.define(ident, {
            ident,
            type: "let",
            pos: stmt.kind.param.pos,
            stmt,
            param: stmt.kind.param,
        });
        return "stop";
    }

    private scoutFnStmts(stmts: Stmt[], syms: Syms) {
        for (const stmt of stmts) {
            if (stmt.kind.type !== "fn") {
                continue;
            }
            if (syms.definedLocally(stmt.kind.ident)) {
                this.reportAlreadyDefined(stmt.kind.ident,
stmt.pos, syms);
                return;
            }
            const ident = stmt.kind.ident;
            syms.define(ident, {
                ident: stmt.kind.ident,
                type: "fn",
                pos: stmt.pos,
                stmt,
            });
        }
    }

    visitFnStmt(stmt: Stmt, syms: Syms): VisitRes {
        if (stmt.kind.type !== "fn") {
            throw new Error("expected fn statement");
```

```
        }
        const fnScopeSyms = new FnSyms(syms);
        for (const param of stmt.kind.params) {
            if (fnScopeSyms.definedLocally(param.ident)) {
                this.reportAlreadyDefined(param.ident, param.pos,
syms);
                continue;
            }
            fnScopeSyms.define(param.ident, {
                ident: param.ident,
                type: "fn_param",
                pos: param.pos,
                param,
            });
        }
        visitExpr(stmt.kind.body, this, fnScopeSyms);
        return "stop";
    }

    visitIdentExpr(expr: Expr, syms: Syms): VisitRes {
        if (expr.kind.type !== "ident") {
            throw new Error("expected ident");
        }
        const ident = expr.kind;
        const symResult = syms.get(ident.value);
        if (!symResult.ok) {
            this.reportUseOfUndefined(ident.value, expr.pos, syms);
            return;
        }
        const sym = symResult.sym;
        expr.kind = {
            type: "sym",
            ident: ident.value,
            sym,
        };
        return "stop";
    }

    visitBlockExpr(expr: Expr, syms: Syms): VisitRes {
        if (expr.kind.type !== "block") {
            throw new Error();
        }
        const childSyms = new LeafSyms(syms);
        this.scoutFnStmts(expr.kind.stmts, childSyms);
        visitStmts(expr.kind.stmts, this, childSyms);
        if (expr.kind.expr) {
            visitExpr(expr.kind.expr, this, childSyms);
        }
        return "stop";
    }

    visitForExpr(expr: Expr, syms: Syms): VisitRes {
        if (expr.kind.type !== "for") {
```

```
            throw new Error();
        }
        const childSyms = new LeafSyms(syms);
        if (expr.kind.decl) visitStmt(expr.kind.decl, this, syms);
        if (expr.kind.cond) visitExpr(expr.kind.cond, this, syms);
        if (expr.kind.incr) visitStmt(expr.kind.incr, this, syms);
        visitExpr(expr.kind.body, this, childSyms);

        return "stop";
    }

    private reportUseOfUndefined(ident: string, pos: Pos, _syms:
Syms) {
        this.reporter.reportError({
            reporter: "Resolver",
            msg: `use of undefined symbol '${ident}'`,
            pos,
        });
        printStackTrace();
    }

    private reportAlreadyDefined(ident: string, pos: Pos, syms:
Syms) {
        this.reporter.reportError({
            reporter: "Resolver",
            msg: `symbol already defined '${ident}'`,
            pos,
        });
        const prev = syms.get(ident);
        if (!prev.ok) {
            throw new Error("expected to be defined");
        }
        if (!prev.sym.pos) {
            return;
        }
        this.reporter.addNote({
            reporter: "Resolver",
            msg: `previous definition of '${ident}'`,
            pos: prev.sym.pos,
        });
        printStackTrace();
    }
}
```

compiler/mod.ts

```
export { compileWithDebug } from "./lib.ts";
```

compiler/ast_visitor.ts

```
import { EType, Expr, Param, Stmt } from "./ast.ts";
```

```
export type VisitRes = "stop" | void;

export interface AstVisitor<Args extends unknown[] = []> {
    visitStmts?(stmts: Stmt[], ...args: Args): VisitRes;
    visitStmt?(stmt: Stmt, ...args: Args): VisitRes;
    visitErrorStmt?(stmt: Stmt, ...args: Args): VisitRes;
    visitImportStmt?(stmt: Stmt, ...args: Args): VisitRes;
    visitBreakStmt?(stmt: Stmt, ...args: Args): VisitRes;
    visitReturnStmt?(stmt: Stmt, ...args: Args): VisitRes;
    visitFnStmt?(stmt: Stmt, ...args: Args): VisitRes;
    visitLetStmt?(stmt: Stmt, ...args: Args): VisitRes;
    visitAssignStmt?(stmt: Stmt, ...args: Args): VisitRes;
    visitExprStmt?(stmt: Stmt, ...args: Args): VisitRes;
    visitExpr?(expr: Expr, ...args: Args): VisitRes;
    visitErrorExpr?(expr: Expr, ...args: Args): VisitRes;
    visitIntExpr?(expr: Expr, ...args: Args): VisitRes;
    visitStringExpr?(expr: Expr, ...args: Args): VisitRes;
    visitIdentExpr?(expr: Expr, ...args: Args): VisitRes;
    visitGroupExpr?(expr: Expr, ...args: Args): VisitRes;
    visitFieldExpr?(expr: Expr, ...args: Args): VisitRes;
    visitIndexExpr?(expr: Expr, ...args: Args): VisitRes;
    visitCallExpr?(expr: Expr, ...args: Args): VisitRes;
    visitUnaryExpr?(expr: Expr, ...args: Args): VisitRes;
    visitBinaryExpr?(expr: Expr, ...args: Args): VisitRes;
    visitIfExpr?(expr: Expr, ...args: Args): VisitRes;
    visitBoolExpr?(expr: Expr, ...args: Args): VisitRes;
    visitNullExpr?(expr: Expr, ...args: Args): VisitRes;
    visitLoopExpr?(expr: Expr, ...args: Args): VisitRes;
    visitWhileExpr?(expr: Expr, ...args: Args): VisitRes;
    visitForInExpr?(expr: Expr, ...args: Args): VisitRes;
    visitForExpr?(expr: Expr, ...args: Args): VisitRes;
    visitBlockExpr?(expr: Expr, ...args: Args): VisitRes;
    visitSymExpr?(expr: Expr, ...args: Args): VisitRes;
    visitParam?(param: Param, ...args: Args): VisitRes;
    visitEType?(etype: EType, ...args: Args): VisitRes;
    visitErrorEType?(etype: EType, ...args: Args): VisitRes;
    visitIdentEType?(etype: EType, ...args: Args): VisitRes;
    visitArrayEType?(etype: EType, ...args: Args): VisitRes;
    visitStructEType?(etype: EType, ...args: Args): VisitRes;
    visitAnno?(etype: EType, ...args: Args): VisitRes;
}

export function visitStmts<Args extends unknown[] = []>(
    stmts: Stmt[],
    v: AstVisitor<Args>,
    ...args: Args
) {
    if (v.visitStmts?.(stmts, ...args) === "stop") return;
    stmts.map((stmt) => visitStmt(stmt, v, ...args));
}

export function visitStmt<Args extends unknown[] = []>(
    stmt: Stmt,
```

```
    v: AstVisitor<Args>,
    ...args: Args
) {
    if (v.visitStmt?.(stmt, ...args) == "stop") return;
    switch (stmt.kind.type) {
        case "error":
            if (v.visitErrorStmt?.(stmt, ...args) == "stop")
return;
            break;
        case "import":
            if (v.visitImportStmt?.(stmt, ...args) == "stop")
return;
            visitExpr(stmt.kind.path, v, ...args);
            break;
        case "break":
            if (v.visitBreakStmt?.(stmt, ...args) == "stop")
return;
            if (stmt.kind.expr) visitExpr(stmt.kind.expr, v,
...args);
            break;
        case "return":
            if (v.visitReturnStmt?.(stmt, ...args) == "stop")
return;
            if (stmt.kind.expr) visitExpr(stmt.kind.expr, v,
...args);
            break;
        case "fn":
            if (v.visitFnStmt?.(stmt, ...args) == "stop") return;
            stmt.kind.params.map((param) => visitParam(param, v,
...args));
            if (stmt.kind.returnType) {
                visitEType(stmt.kind.returnType, v, ...args);
            }
            visitExpr(stmt.kind.body, v, ...args);
            break;
        case "let":
            if (v.visitLetStmt?.(stmt, ...args) == "stop") return;
            visitParam(stmt.kind.param, v, ...args);
            visitExpr(stmt.kind.value, v, ...args);
            break;
        case "assign":
            if (v.visitAssignStmt?.(stmt, ...args) == "stop")
return;
            visitExpr(stmt.kind.subject, v, ...args);
            visitExpr(stmt.kind.value, v, ...args);
            break;
        case "expr":
            if (v.visitExprStmt?.(stmt, ...args) == "stop") return;
            visitExpr(stmt.kind.expr, v, ...args);
            break;
    }
}
```

```
export function visitExpr<Args extends unknown[] = []>(
    expr: Expr,
    v: AstVisitor<Args>,
    ...args: Args
) {
    if (v.visitExpr?.(expr, ...args) == "stop") return;
    switch (expr.kind.type) {
        case "error":
            if (v.visitErrorExpr?.(expr, ...args) == "stop")
return;
            break;
        case "string":
            if (v.visitStringExpr?.(expr, ...args) == "stop")
return;
            break;
        case "int":
            if (v.visitIntExpr?.(expr, ...args) == "stop") return;
            break;
        case "ident":
            if (v.visitIdentExpr?.(expr, ...args) == "stop")
return;
            break;
        case "group":
            if (v.visitGroupExpr?.(expr, ...args) == "stop")
return;
            visitExpr(expr.kind.expr, v, ...args);
            break;
        case "field":
            if (v.visitFieldExpr?.(expr, ...args) == "stop")
return;
            visitExpr(expr.kind.subject, v, ...args);
            break;
        case "index":
            if (v.visitIndexExpr?.(expr, ...args) == "stop")
return;
            visitExpr(expr.kind.subject, v, ...args);
            visitExpr(expr.kind.value, v, ...args);
            break;
        case "call":
            if (v.visitCallExpr?.(expr, ...args) == "stop") return;
            visitExpr(expr.kind.subject, v, ...args);
            expr.kind.args.map((arg) => visitExpr(arg, v,
...args));
            break;
        case "unary":
            if (v.visitUnaryExpr?.(expr, ...args) == "stop")
return;
            visitExpr(expr.kind.subject, v, ...args);
            break;
        case "binary":
            if (v.visitBinaryExpr?.(expr, ...args) == "stop")
return;
            visitExpr(expr.kind.left, v, ...args);
```

```
            visitExpr(expr.kind.right, v, ...args);
            break;
        case "if":
            if (v.visitIfExpr?.(expr, ...args) == "stop") return;
            visitExpr(expr.kind.cond, v, ...args);
            visitExpr(expr.kind.truthy, v, ...args);
            if (expr.kind.falsy) visitExpr(expr.kind.falsy, v,
...args);
            break;
        case "bool":
            if (v.visitBoolExpr?.(expr, ...args) == "stop") return;
            break;
        case "null":
            if (v.visitNullExpr?.(expr, ...args) == "stop") return;
            break;
        case "loop":
            if (v.visitLoopExpr?.(expr, ...args) == "stop") return;
            visitExpr(expr.kind.body, v, ...args);
            break;
        case "while":
            if (v.visitWhileExpr?.(expr, ...args) == "stop")
return;
            visitExpr(expr.kind.cond, v, ...args);
            visitExpr(expr.kind.body, v, ...args);
            break;
        case "for_in":
            if (v.visitForInExpr?.(expr, ...args) == "stop")
return;
            visitParam(expr.kind.param, v, ...args);
            visitExpr(expr.kind.value, v, ...args);
            visitExpr(expr.kind.body, v, ...args);
            break;
        case "for":
            if (v.visitForExpr?.(expr, ...args) == "stop") return;
            if (expr.kind.decl) visitStmt(expr.kind.decl, v,
...args);
            if (expr.kind.cond) visitExpr(expr.kind.cond, v,
...args);
            if (expr.kind.incr) visitStmt(expr.kind.incr, v,
...args);
            visitExpr(expr.kind.body, v, ...args);
            break;
        case "block":
            if (v.visitBlockExpr?.(expr, ...args) == "stop")
return;
            expr.kind.stmts.map((stmt) => visitStmt(stmt, v,
...args));
            if (expr.kind.expr) visitExpr(expr.kind.expr, v,
...args);
            break;
        case "sym":
            if (v.visitSymExpr?.(expr, ...args) == "stop") return;
            break;
```

```
    }
}

export function visitParam<Args extends unknown[] = []>(
    param: Param,
    v: AstVisitor<Args>,
    ...args: Args
) {
    if (v.visitParam?.(param, ...args) == "stop") return;
    if (param.etype) visitEType(param.etype, v, ...args);
}

export function visitEType<Args extends unknown[] = []>(
    etype: EType,
    v: AstVisitor<Args>,
    ...args: Args
) {
    if (v.visitEType?.(etype, ...args) == "stop") return;
    switch (etype.kind.type) {
        case "error":
            if (v.visitErrorEType?.(etype, ...args) == "stop")
return;
            break;
        case "ident":
            if (v.visitIdentEType?.(etype, ...args) == "stop")
return;
            break;
        case "array":
            if (v.visitArrayEType?.(etype, ...args) == "stop")
return;
            if (etype.kind.inner) visitEType(etype.kind.inner, v,
...args);
            break;
        case "struct":
            if (v.visitStructEType?.(etype, ...args) == "stop")
return;
            etype.kind.fields.map((field) => visitParam(field, v,
...args));
            break;
    }
}

export function stmtToString(stmt: Stmt): string {
    const body = (() => {
        switch (stmt.kind.type) {
            case "assign":
                return `{ subject:
${exprToString(stmt.kind.subject)}, value: ${
                    exprToString(stmt.kind.value)
                } }`;
        }
        return "(<not implemented>)";
    })();
```

```
    const { line } = stmt.pos;
    return `${stmt.kind.type}:${line}${body}`;
}

export function exprToString(expr: Expr): string {
    const body = (() => {
        switch (expr.kind.type) {
            case "binary":
                return `(${
                    exprToString(expr.kind.left)
                } ${expr.kind.binaryType}
${exprToString(expr.kind.right)})`;
            case "sym":
                return `(${expr.kind.ident})`;
        }
        return "(<not implemented>)";
    })();
    const { line } = expr.pos;
    return `${expr.kind.type}:${line}${body}`;
}
```

compiler/lexer.ts

```
import { Reporter } from "./info.ts";
import { Pos, Token } from "./token.ts";

export class Lexer {
    private index = 0;
    private line = 1;
    private col = 1;

    public constructor(private text: string, private reporter:
Reporter) {}

    public next(): Token | null {
        if (this.done()) {
            return null;
        }
        const pos = this.pos();
        if (this.test(/[ \t\n\r]/)) {
            while (!this.done() && this.test(/[ \t\n\r]/)) {
                this.step();
            }
            return this.next();
        }

        if (this.test(/[a-zA-Z_]/)) {
            let value = "";
            while (!this.done() && this.test(/[a-zA-Z0-9_]/)) {
                value += this.current();
                this.step();
            }
            const keywords = [
```

```
                "break",
                "return",
                "let",
                "fn",
                "loop",
                "if",
                "else",
                "struct",
                "import",
                "false",
                "true",
                "null",
                "or",
                "and",
                "not",
                "while",
                "for",
                "in",
            ];
            if (keywords.includes(value)) {
                return this.token(value, pos);
            } else {
                return { ...this.token("ident", pos), identValue:
value };
            }
        }
        if (this.test(/[1-9]/)) {
            let textValue = "";
            while (!this.done() && this.test(/[0-9]/)) {
                textValue += this.current();
                this.step();
            }
            return { ...this.token("int", pos), intValue:
parseInt(textValue) };
        }

        if (this.test("0")) {
            this.step();
            if (!this.done() && this.test(/[0-9]/)) {
                this.report("invalid number", pos);
                return this.token("error", pos);
            }
            return { ...this.token("int", pos), intValue: 0 };
        }

        if (this.test('"')) {
            this.step();
            let value = "";
            while (!this.done() && !this.test('"')) {
                if (this.test("\\")) {
                    this.step();
                    if (this.done()) {
                        break;
```

```
                }
                value += {
                    n: "\n",
                    t: "\t",
                    "0": "\0",
                }[this.current()] ?? this.current();
            } else {
                value += this.current();
            }
            this.step();
        }
        if (this.done() || !this.test('"')) {
            this.report("unclosed/malformed string", pos);
            return this.token("error", pos);
        }
        this.step();
        return { ...this.token("string", pos), stringValue:
value };
    }
    if (this.test(/[\+\{\};=\-\*\(\)\.,:;\[\]><!0#]/)) {
        const first = this.current();
        this.step();
        if (first === "=" && !this.done() && this.test("=")) {
            this.step();
            return this.token("==", pos);
        }
        if (first === "<" && !this.done() && this.test("=")) {
            this.step();
            return this.token("<=", pos);
        }
        if (first === ">" && !this.done() && this.test("=")) {
            this.step();
            return this.token(">=", pos);
        }
        if (first === "-" && !this.done()) {
            if (this.test(">")) {
                this.step();
                return this.token("->", pos);
            }
            if (this.test("=")) {
                this.step();
                return this.token("-=", pos);
            }
        }
        if (first === "!" && !this.done() && this.test("=")) {
            this.step();
            return this.token("!=", pos);
        }
        if (first === "+" && !this.done() && this.test("=")) {
            this.step();
            return this.token("+=", pos);
        }
        if (first === ":") {
```

```
            if (!this.done() && this.test(":")) {
                this.step();
                if (!this.done() && this.test("<")) {
                    this.step();
                    return this.token("::<", pos);
                }
                return this.token("::", pos);
            }
        }
        return this.token(first, pos);
    }
    if (this.test("/")) {
        this.step();
        if (this.test("/")) {
            while (!this.done() && !this.test("\n")) {
                this.step();
            }
            return this.next();
        }
        return this.token("/", pos);
    }

    this.report(`illegal character '${this.current()}'`, pos);
    this.step();
    return this.next();
}

private done(): boolean {
    return this.index >= this.text.length;
}

private current(): string {
    return this.text[this.index];
}

public currentPos(): Pos {
    return this.pos();
}

private step() {
    if (this.done()) {
        return;
    }
    if (this.current() === "\n") {
        this.line += 1;
        this.col = 1;
    } else {
        this.col += 1;
    }
    this.index += 1;
}

private pos(): Pos {
```

```
        return {
            index: this.index,
            line: this.line,
            col: this.col,
        };
    }

    private token(type: string, pos: Pos): Token {
        const length = this.index - pos.index;
        return { type, pos, length };
    }

    private test(pattern: RegExp | string): boolean {
        if (typeof pattern === "string") {
            return this.current() === pattern;
        } else {
            return pattern.test(this.current());
        }
    }

    private report(msg: string, pos: Pos) {
        this.reporter.reportError({
            msg,
            pos,
            reporter: "Lexer",
        });
    }
}
```

compiler/token.ts

```
export type Token = {
    type: string;
    pos: Pos;
    length: number;
    identValue?: string;
    intValue?: number;
    stringValue?: string;
};

export type Pos = {
    index: number;
    line: number;
    col: number;
};
```

compiler/vtype.ts

```
export type VType =
    | { type: "error" }
    | { type: "unknown" }
    | { type: "null" }
    | { type: "int" }
```

```
    | { type: "string" }
    | { type: "bool" }
    | { type: "array"; inner: VType }
    | { type: "struct"; fields: VTypeParam[] }
    | { type: "fn"; params: VTypeParam[]; returnType: VType };

export type VTypeParam = {
    ident: string;
    vtype: VType;
};

export function vtypesEqual(a: VType, b: VType): boolean {
    if (a.type !== b.type) {
        return false;
    }
    if (
        ["error", "unknown", "null", "int", "string", "bool"]
            .includes(a.type)
    ) {
        return true;
    }
    if (a.type === "array" && b.type === "array") {
        return vtypesEqual(a.inner, b.inner);
    }
    if (a.type === "fn" && b.type === "fn") {
        if (a.params.length !== b.params.length) {
            return false;
        }
        for (let i = 0; i < a.params.length; ++i) {
            if (!vtypesEqual(a.params[i].vtype, b.params[i].vtype))
{
                return false;
            }
        }
        return vtypesEqual(a.returnType, b.returnType);
    }
    return false;
}

export function vtypeToString(vtype: VType): string {
    if (
        ["error", "unknown", "null", "int", "string", "bool"]
            .includes(vtype.type)
    ) {
        return vtype.type;
    }
    if (vtype.type === "array") {
        return `[${vtypeToString(vtype.inner)}]`;
    }
    if (vtype.type === "fn") {
        const paramString = vtype.params.map((param) =>
            `${param.ident}: ${vtypeToString(param.vtype)}`
        )
```

```
            .join(", ");
        return `fn (${paramString}) ->
${vtypeToString(vtype.returnType)}`;
    }
    throw new Error(`unhandled vtype '${vtype.type}'`);
}
```

compiler/checker.ts

```
import { EType, Expr, Stmt } from "./ast.ts";
import { printStackTrace, Reporter } from "./info.ts";
import { Pos } from "./token.ts";
import { VType, VTypeParam, vtypesEqual, vtypeToString } from "./
vtype.ts";

export class Checker {
    private fnReturnStack: VType[] = [];
    private loopBreakStack: VType[][] = [];

    public constructor(private reporter: Reporter) {}

    public check(stmts: Stmt[]) {
        this.checkFnHeaders(stmts);
        for (const stmt of stmts) {
            this.checkStmt(stmt);
        }
    }

    private checkFnHeaders(stmts: Stmt[]) {
        for (const stmt of stmts) {
            if (stmt.kind.type !== "fn") {
                continue;
            }
            const returnType: VType = stmt.kind.returnType
                ? this.checkEType(stmt.kind.returnType)
                : { type: "null" };
            const params: VTypeParam[] = [];
            for (const param of stmt.kind.params) {
                if (param.etype === undefined) {
                    this.report("parameter types must be defined",
param.pos);
                    stmt.kind.vtype = { type: "error" };
                }
                const vtype = this.checkEType(param.etype!);
                param.vtype = vtype;
                params.push({ ident: param.ident, vtype });
            }
            stmt.kind.vtype = { type: "fn", params, returnType };
        }
    }

    public checkStmt(stmt: Stmt) {
        switch (stmt.kind.type) {
```

```
            case "error":
                return { type: "error" };
            case "break":
                return this.checkBreakStmt(stmt);
            case "return":
                return this.checkReturnStmt(stmt);
            case "fn":
                return this.checkFnStmt(stmt);
            case "let":
                return this.checkLetStmt(stmt);
            case "assign":
                return this.checkAssignStmt(stmt);
            case "expr":
                return this.checkExpr(stmt.kind.expr);
        }
    }

    public checkBreakStmt(stmt: Stmt) {
        if (stmt.kind.type !== "break") {
            throw new Error();
        }
        const pos = stmt.pos;
        if (this.loopBreakStack.length === 0) {
            this.report("cannot break outside loop context", pos);
            return;
        }
        const exprType: VType = stmt.kind.expr
            ? this.checkExpr(stmt.kind.expr)
            : { type: "null" };
        const breakTypes = this.loopBreakStack.at(-1)!;
        if (breakTypes.length === 0) {
            breakTypes.push(exprType);
            return;
        }
        const prevBreakType = breakTypes.at(-1)!;
        if (!vtypesEqual(prevBreakType, exprType)) {
            this.report(
                `incompatible types for break` +
                    `, got ${exprType}` +
                    ` incompatible with ${prevBreakType}`,
                pos,
            );
            return;
        }
        breakTypes.push(exprType);
    }

    public checkReturnStmt(stmt: Stmt) {
        if (stmt.kind.type !== "return") {
            throw new Error();
        }
        const pos = stmt.pos;
        if (this.fnReturnStack.length === 0) {
```

```
            this.report("cannot return outside fn context", pos);
            return;
        }
        const exprType: VType = stmt.kind.expr
            ? this.checkExpr(stmt.kind.expr)
            : { type: "null" };
        const returnType = this.fnReturnStack.at(-1)!;
        if (!vtypesEqual(exprType, returnType)) {
            this.report(
                `incompatible return type` +
                    `, got ${exprType}` +
                    `, expected ${returnType}`,
                pos,
            );
        }
    }

    public checkFnStmt(stmt: Stmt) {
        if (stmt.kind.type !== "fn") {
            throw new Error();
        }
        const pos = stmt.pos;
        if (stmt.kind.vtype!.type !== "fn") {
            throw new Error();
        }

        if (
            stmt.kind.anno?.ident === "remainder" ||
            stmt.kind.anno?.ident === "builtin"
        ) {
            return;
        }

        const { returnType } = stmt.kind.vtype!;
        this.fnReturnStack.push(returnType);
        const body = this.checkExpr(stmt.kind.body);
        this.fnReturnStack.pop();

        if (!vtypesEqual(returnType, body)) {
            this.report(
                `incompatible return type` +
                    `, expected '${vtypeToString(returnType)}'` +
                    `, got '${vtypeToString(body)}'`,
                pos,
            );
        }
    }

    public checkLetStmt(stmt: Stmt) {
        if (stmt.kind.type !== "let") {
            throw new Error();
        }
        const pos = stmt.pos;
```

```
        const value = this.checkExpr(stmt.kind.value);
        if (stmt.kind.param.etype) {
            const paramVtype =
this.checkEType(stmt.kind.param.etype);
            if (!vtypesEqual(value, paramVtype)) {
                this.report(
                    `incompatible value type` +
                        `, got '${vtypeToString(value)}'` +
                        `, expected
'${vtypeToString(paramVtype)}'`,
                    pos,
                );
                return;
            }
        }
        stmt.kind.param.vtype = value;
    }

    public checkAssignStmt(stmt: Stmt) {
        if (stmt.kind.type !== "assign") {
            throw new Error();
        }
        const pos = stmt.pos;
        if (stmt.kind.assignType !== "=") {
            throw new Error("invalid ast: compound assign should be
desugered");
        }
        const value = this.checkExpr(stmt.kind.value);
        switch (stmt.kind.subject.kind.type) {
            case "field": {
                const subject =
this.checkExpr(stmt.kind.subject.kind.subject);
                if (subject.type !== "struct") {
                    this.report("cannot use field on non-struct",
pos);
                    return { type: "error" };
                }
                const fieldValue = stmt.kind.subject.kind.value;
                const found = subject.fields.find((param) =>
                    param.ident === fieldValue
                );
                if (!found) {
                    this.report(
                        `no field named
'${stmt.kind.subject.kind.value}' on struct`,
                        pos,
                    );
                    return { type: "error" };
                }
                if (!vtypesEqual(found.vtype, value)) {
                    this.report(
                        `cannot assign incompatible type to field
'${found.ident}'` +
```

```
                              `, got '${vtypeToString(value)}'` +
                              `, expected
'${vtypeToString(found.vtype)}'`,
                          pos,
                      );
                      return;
                  }
                  return;
              }
              case "index": {
                  const subject =
this.checkExpr(stmt.kind.subject.kind.subject);
                  if (subject.type !== "array" && subject.type !==
"string") {
                      this.report(
                          `cannot index on non-array, got:
${subject.type}`,
                          pos,
                      );
                      return { type: "error" };
                  }
                  const indexValue =
this.checkExpr(stmt.kind.subject.kind.value);
                  if (indexValue.type !== "int") {
                      this.report("cannot index on array with non-
int", pos);
                      return { type: "error" };
                  }
                  if (
                      subject.type == "array" &&
                      !vtypesEqual(subject.inner, value)
                  ) {
                      this.report(
                          `cannot assign incompatible type to array `
+
                              `'${vtypeToString(subject)}'` +
                              `, got '${vtypeToString(value)}'`,
                          pos,
                      );
                      return;
                  }
                  return;
              }
              case "sym": {
                  if (stmt.kind.subject.kind.sym.type !== "let") {
                      this.report("cannot only assign to let-symbol",
pos);
                      return { type: "error" };
                  }
                  if (
                      !
vtypesEqual(stmt.kind.subject.kind.sym.param.vtype!, value)
                  ) {
```

```
                        this.report(
                            `cannot assign to incompatible type` +
                                `, got '${vtypeToString(value)}'` +
                                `, expected '${
                                    vtypeToString(

stmt.kind.subject.kind.sym.param.vtype!,
                                    )
                                }'`,
                            pos,
                        );
                        return;
                    }
                    return;
                }
                default:
                    this.report("unassignable expression", pos);
                    return;
            }
        }

    public checkExpr(expr: Expr): VType {
        const vtype = ((): VType => {
            switch (expr.kind.type) {
                case "error":
                    throw new Error("error in AST");
                case "ident":
                    throw new Error("ident expr in AST");
                case "sym":
                    return this.checkSymExpr(expr);
                case "null":
                    return { type: "null" };
                case "int":
                    return { type: "int" };
                case "bool":
                    return { type: "bool" };
                case "string":
                    return { type: "string" };
                case "group":
                    return this.checkExpr(expr.kind.expr);
                case "field":
                    return this.checkFieldExpr(expr);
                case "index":
                    return this.checkIndexExpr(expr);
                case "call":
                    return this.checkCallExpr(expr);
                case "unary":
                    return this.checkUnaryExpr(expr);
                case "binary":
                    return this.checkBinaryExpr(expr);
                case "if":
                    return this.checkIfExpr(expr);
                case "loop":
```

```
                return this.checkLoopExpr(expr);
            case "while":
            case "for_in":
            case "for":
                throw new Error(
                    "invalid ast: special loops should be
desugered",
                );
            case "block":
                return this.checkBlockExpr(expr);
        }
        // throw new Error(`unhandled type ${expr.kind.type}`);
    })();
    return expr.vtype = vtype;
}

public checkSymExpr(expr: Expr): VType {
    if (expr.kind.type !== "sym") {
        throw new Error();
    }
    switch (expr.kind.sym.type) {
        case "let":
            return expr.kind.sym.param.vtype!;
        case "fn": {
            const fnStmt = expr.kind.sym.stmt!;
            if (fnStmt.kind.type !== "fn") {
                throw new Error();
            }
            const vtype = fnStmt.kind.vtype!;
            if (vtype.type !== "fn") {
                throw new Error();
            }
            const { params, returnType } = vtype;
            return { type: "fn", params, returnType };
        }
        case "fn_param":
            return expr.kind.sym.param.vtype!;
        case "builtin":
        case "let_static":
        case "closure":
            throw new Error(
                `not implemented, sym type
'${expr.kind.sym.type}'`,
            );
    }
}

public checkFieldExpr(expr: Expr): VType {
    if (expr.kind.type !== "field") {
        throw new Error();
    }
    const pos = expr.pos;
    const subject = this.checkExpr(expr.kind.subject);
```

```
        if (subject.type !== "struct") {
            this.report("cannot use field on non-struct", pos);
            return { type: "error" };
        }
        const value = expr.kind.value;
        const found = subject.fields.find((param) => param.ident
=== value);
        if (!found) {
            this.report(
                `no field named '${expr.kind.value}' on struct`,
                pos,
            );
            return { type: "error" };
        }
        return found.vtype;
    }

    public checkIndexExpr(expr: Expr): VType {
        if (expr.kind.type !== "index") {
            throw new Error();
        }
        const pos = expr.pos;
        const subject = this.checkExpr(expr.kind.subject);
        if (subject.type !== "array" && subject.type !== "string")
{
            this.report(`cannot index on non-array, got:
${subject.type}`, pos);
            return { type: "error" };
        }
        const value = this.checkExpr(expr.kind.value);
        if (value.type !== "int") {
            this.report("cannot index on array with non-int", pos);
            return { type: "error" };
        }
        if (subject.type === "array") {
            return subject.inner;
        }
        return { type: "int" };
    }

    public checkCallExpr(expr: Expr): VType {
        if (expr.kind.type !== "call") {
            throw new Error();
        }
        const pos = expr.pos;
        const subject = this.checkExpr(expr.kind.subject);
        if (subject.type !== "fn") {
            this.report("cannot call non-fn", pos);
            return { type: "error" };
        }
        const args = expr.kind.args.map((arg) =>
this.checkExpr(arg));
        if (args.length !== subject.params.length) {
```

```
            this.report(
                `incorrect number of arguments` +
                    `, expected ${subject.params.length}`,
                pos,
            );
        }
        for (let i = 0; i < args.length; ++i) {
            if (!vtypesEqual(args[i], subject.params[i].vtype)) {
                this.report(
                    `incorrect argument ${i}
'${subject.params[i].ident}'` +
                        `, expected
${vtypeToString(subject.params[i].vtype)}` +
                        `, got ${vtypeToString(args[i])}`,
                    pos,
                );
                break;
            }
        }
        return subject.returnType;
    }

    public checkUnaryExpr(expr: Expr): VType {
        if (expr.kind.type !== "unary") {
            throw new Error();
        }
        const pos = expr.pos;
        const subject = this.checkExpr(expr.kind.subject);
        for (const operation of simpleUnaryOperations) {
            if (operation.unaryType !== expr.kind.unaryType) {
                continue;
            }
            if (!vtypesEqual(operation.operand, subject)) {
                continue;
            }
            return operation.result ?? operation.operand;
        }
        this.report(
            `cannot apply unary operation '${expr.kind.unaryType}'
` +
                `on type '${vtypeToString(subject)}'`,
            pos,
        );
        return { type: "error" };
    }

    public checkBinaryExpr(expr: Expr): VType {
        if (expr.kind.type !== "binary") {
            throw new Error();
        }
        const pos = expr.pos;
        const left = this.checkExpr(expr.kind.left);
        const right = this.checkExpr(expr.kind.right);
```

```
        for (const operation of simpleBinaryOperations) {
            if (operation.binaryType !== expr.kind.binaryType) {
                continue;
            }
            if (!vtypesEqual(operation.operand, left)) {
                continue;
            }
            if (!vtypesEqual(left, right)) {
                continue;
            }
            return operation.result ?? operation.operand;
        }
        this.report(
            `cannot apply binary operation
'${expr.kind.binaryType}' ` +
                `on types '${vtypeToString(left)}' and '${
                    vtypeToString(right)
                }'`,
            pos,
        );
        return { type: "error" };
    }

    public checkIfExpr(expr: Expr): VType {
        if (expr.kind.type !== "if") {
            throw new Error();
        }
        const pos = expr.pos;
        const cond = this.checkExpr(expr.kind.cond);
        const truthy = this.checkExpr(expr.kind.truthy);
        const falsy = expr.kind.falsy
            ? this.checkExpr(expr.kind.falsy)
            : undefined;
        if (cond.type !== "bool") {
            this.report(
                `if condition should be 'bool', got
'${vtypeToString(cond)}'`,
                pos,
            );
            return { type: "error" };
        }
        if (falsy === undefined && truthy.type !== "null") {
            this.report(
                `if expressions without false-case must result in
type 'null'` +
                    `, got '${vtypeToString(truthy)}'`,
                pos,
            );
            return { type: "error" };
        }
        if (falsy !== undefined && !vtypesEqual(truthy, falsy)) {
            this.report(
                `if cases must be compatible, got incompatible
```

```
types` +
                    ` '${vtypeToString(truthy)}'` +
                    ` and '${vtypeToString(falsy)}'`,
                pos,
            );
            return { type: "error" };
        }
        return truthy;
    }

    public checkLoopExpr(expr: Expr): VType {
        if (expr.kind.type !== "loop") {
            throw new Error();
        }
        const pos = expr.pos;
        this.loopBreakStack.push([]);
        const body = this.checkExpr(expr.kind.body);
        if (body.type !== "null") {
            this.report(
                `loop body must result in type 'null'` +
                    `, got '${vtypeToString(body)}'`,
                pos,
            );
            return { type: "error" };
        }
        const loopBreakTypes = this.loopBreakStack.pop()!;
        if (loopBreakTypes.length === 0) {
            return { type: "null" };
        }
        const breakType = loopBreakTypes.reduce<[VType, boolean,
VType]>(
            (acc, curr) => {
                const [resulting, isIncompatible, outlier] = acc;
                if (isIncompatible) {
                    return acc;
                }
                if (!vtypesEqual(resulting, curr)) {
                    return [resulting, true, curr];
                }
                return [resulting, false, outlier];
            },
            [{ type: "null" }, false, { type: "null" }],
        );
        if (breakType[1]) {
            this.report(
                `incompatible types in break statements` +
                    `, got '${vtypeToString(breakType[2])}'` +
                    ` incompatible with
${vtypeToString(breakType[0])}`,
                pos,
            );
            return { type: "error" };
        }
```

```
        return breakType[0];
    }

    public checkBlockExpr(expr: Expr): VType {
        if (expr.kind.type !== "block") {
            throw new Error();
        }
        this.checkFnHeaders(expr.kind.stmts);
        for (const stmt of expr.kind.stmts) {
            this.checkStmt(stmt);
        }
        return expr.kind.expr
            ? this.checkExpr(expr.kind.expr)
            : { type: "null" };
    }

    public checkEType(etype: EType): VType {
        const pos = etype.pos;
        if (etype.kind.type === "ident") {
            if (etype.kind.value === "null") {
                return { type: "null" };
            }
            if (etype.kind.value === "int") {
                return { type: "int" };
            }
            if (etype.kind.value === "bool") {
                return { type: "bool" };
            }
            if (etype.kind.value === "string") {
                return { type: "string" };
            }
            this.report(`undefined type '${etype.kind.value}'`,
pos);
            return { type: "error" };
        }
        if (etype.kind.type === "array") {
            const inner = this.checkEType(etype.kind.inner);
            return { type: "array", inner };
        }
        if (etype.kind.type === "struct") {
            const noTypeTest = etype.kind.fields.reduce(
                (acc, param) => [acc[0] || !param.etype,
param.ident],
                [false, ""],
            );
            if (noTypeTest[0]) {
                this.report(
                    `field '${noTypeTest[1]}' declared without
type`,
                    pos,
                );
                return { type: "error" };
            }
```

```
            const declaredTwiceTest = etype.kind.fields.reduce<
                [boolean, string[], string]
            >(
                (acc, curr) => {
                    if (acc[0]) {
                        return acc;
                    }
                    if (acc[1].includes(curr.ident)) {
                        return [true, acc[1], curr.ident];
                    }
                    return [false, [...acc[1], curr.ident], ""];
                },
                [false, [], ""],
            );
            if (
                declaredTwiceTest[0]
            ) {
                this.report(`field ${declaredTwiceTest[2]} defined
twice`, pos);
                return { type: "error" };
            }
            const fields = etype.kind.fields.map((param):
VTypeParam => ({
                ident: param.ident,
                vtype: this.checkEType(param.etype!),
            }));
            return { type: "struct", fields };
        }
        throw new Error(`unknown explicit type ${etype.kind.type}
`);
    }

    private report(msg: string, pos: Pos) {
        this.reporter.reportError({ reporter: "Checker", msg, pos
});
        printStackTrace();
    }
}

const simpleUnaryOperations: {
    unaryType: string;
    operand: VType;
    result?: VType;
}[] = [
    { unaryType: "not", operand: { type: "bool" } },
    { unaryType: "-", operand: { type: "int" } },
];

const simpleBinaryOperations: {
    binaryType: string;
    operand: VType;
    result?: VType;
}[] = [
```

```
    // arithmetic
    { binaryType: "+", operand: { type: "int" } },
    { binaryType: "+", operand: { type: "string" } },
    { binaryType: "-", operand: { type: "int" } },
    { binaryType: "*", operand: { type: "int" } },
    { binaryType: "/", operand: { type: "int" } },
    // logical
    { binaryType: "and", operand: { type: "bool" } },
    { binaryType: "or", operand: { type: "bool" } },
    // equality
    { binaryType: "==", operand: { type: "null" }, result: { type:
"bool" } },
    { binaryType: "==", operand: { type: "int" }, result: { type:
"bool" } },
    { binaryType: "==", operand: { type: "string" }, result: {
type: "bool" } },
    { binaryType: "==", operand: { type: "bool" }, result: { type:
"bool" } },
    { binaryType: "!=", operand: { type: "null" }, result: { type:
"bool" } },
    { binaryType: "!=", operand: { type: "int" }, result: { type:
"bool" } },
    { binaryType: "!=", operand: { type: "string" }, result: {
type: "bool" } },
    { binaryType: "!=", operand: { type: "bool" }, result: { type:
"bool" } },
    // comparison
    { binaryType: "<", operand: { type: "int" }, result: { type:
"bool" } },
    { binaryType: ">", operand: { type: "int" }, result: { type:
"bool" } },
    { binaryType: "<=", operand: { type: "int" }, result: { type:
"bool" } },
    { binaryType: ">=", operand: { type: "int" }, result: { type:
"bool" } },
];
```

compiler/resolver_syms.ts

```
import { Sym } from "./ast.ts";

export type SymMap = { [ident: string]: Sym };

export interface Syms {
    define(ident: string, sym: Sym): void;
    definedLocally(ident: string): boolean;
    get(ident: string): { ok: true; sym: Sym } | { ok: false };
}

export class GlobalSyms implements Syms {
    private syms: SymMap = {};

    public constructor() {}
```

```
    public define(ident: string, sym: Sym) {
        if (sym.type === "let") {
            this.define(ident, {
                ...sym,
                type: "let_static",
            });
            return;
        }
        this.syms[ident] = sym;
    }

    public definedLocally(ident: string): boolean {
        return ident in this.syms;
    }

    public get(ident: string): { ok: true; sym: Sym } | { ok: false
} {
        if (ident in this.syms) {
            return { ok: true, sym: this.syms[ident] };
        }
        return { ok: false };
    }
}

export class StaticSyms implements Syms {
    private syms: SymMap = {};

    public constructor(private parent: GlobalSyms) {}

    public define(ident: string, sym: Sym) {
        if (sym.type === "let") {
            this.define(ident, {
                ...sym,
                type: "let_static",
            });
            return;
        }
        this.syms[ident] = sym;
    }

    public definedLocally(ident: string): boolean {
        return ident in this.syms;
    }

    public get(ident: string): { ok: true; sym: Sym } | { ok: false
} {
        if (ident in this.syms) {
            return { ok: true, sym: this.syms[ident] };
        }
        return this.parent.get(ident);
    }
}
```

```
export class FnSyms implements Syms {
    private syms: SymMap = {};

    public constructor(private parent: Syms) {}

    public define(ident: string, sym: Sym) {
        if (sym.type === "let") {
            this.define(ident, {
                ...sym,
                type: "closure",
                inner: sym,
            });
            return;
        }
        this.syms[ident] = sym;
    }

    public definedLocally(ident: string): boolean {
        return ident in this.syms;
    }

    public get(ident: string): { ok: true; sym: Sym } | { ok: false
} {
        if (ident in this.syms) {
            return { ok: true, sym: this.syms[ident] };
        }
        return this.parent.get(ident);
    }
}

export class LeafSyms implements Syms {
    private syms: SymMap = {};

    public constructor(private parent: Syms) {}

    public define(ident: string, sym: Sym) {
        this.syms[ident] = sym;
    }

    public definedLocally(ident: string): boolean {
        return ident in this.syms;
    }

    public get(ident: string): { ok: true; sym: Sym } | { ok: false
} {
        if (ident in this.syms) {
            return { ok: true, sym: this.syms[ident] };
        }
        return this.parent.get(ident);
    }
}
```

compiler/architecture.txt

```
Mnemonic        Arg     Arg description
----------------------------------------
Nop
PushNull
PushInt         int     initial value
PushString      string  initial value
PushArray
PushStruct
PushPtr         ptr     pointer value
Pop
LoadLocal       int     stack position
StoreLocal      int     stack position
Call            int     arg count
Return
Jump
JumpIfNotZero
Add
Subtract
Multiply
Divide
Remainder
Equal
LessThan
And
Or
Xor
Not
```

compiler/assembler.ts

```
import { opToString } from "./arch.ts";

export type Line = { labels?: string[]; ins: Ins };

export type Ins = Lit[];

export type Label = { label: string };

export type Lit = number | string | boolean | Label;

export type Locs = { [key: string]: number };
export type Refs = { [key: number]: string };

export class Assembler {
```

```
private lines: Line[] = [];
private addedLabels: string[] = [];

private constructor(private labelCounter: number) {}

public static newRoot(): Assembler {
    return new Assembler(0);
}

public fork(): Assembler {
    return new Assembler(this.labelCounter);
}

public join(assembler: Assembler) {
    this.labelCounter = assembler.labelCounter;
    if (assembler.lines.length < 0) {
        return;
    }
    if (assembler.lines[0].labels !== undefined) {
        this.addedLabels.push(...assembler.lines[0].labels);
    }
    this.add(...assembler.lines[0].ins);
    this.lines.push(...assembler.lines.slice(1));
}

public add(...ins: Ins): Assembler {
    if (this.addedLabels.length > 0) {
        this.lines.push({ ins, labels: this.addedLabels });
        this.addedLabels = [];
        return this;
    }
    this.lines.push({ ins });
    return this;
}

public makeLabel(): Label {
    return { label: `.L${(this.labelCounter++).toString()}` };
}

public setLabel({ label }: Label) {
    this.addedLabels.push(label);
}

public assemble(): { program: number[]; locs: Locs } {
    let ip = 0;
    const program: number[] = [];
    const locs: Locs = {};
    const refs: Refs = {};

    let selectedLabel = "";
    for (const line of this.lines) {
        for (const label of line.labels ?? []) {
            const isAbsLabel = !label.startsWith(".");
```

```
                if (isAbsLabel) {
                    selectedLabel = label;
                    locs[label] = ip;
                } else {
                    locs[`${selectedLabel}${label}`] = ip;
                }
            }
            for (const lit of line.ins as Lit[]) {
                if (typeof lit === "number") {
                    program.push(lit);
                    ip += 1;
                } else if (typeof lit === "boolean") {
                    program.push(lit ? 1 : 0);
                    ip += 1;
                } else if (typeof lit === "string") {
                    program.push(lit.length);
                    ip += 1;
                    for (let i = 0; i < lit.length; ++i) {
                        program.push(lit.charCodeAt(i));
                        ip += 1;
                    }
                } else {
                    program.push(0);
                    refs[ip] = lit.label.startsWith(".")
                        ? `${selectedLabel}${lit.label}`
                        : refs[ip] = lit.label;
                    ip += 1;
                }
            }
        }
        for (let i = 0; i < program.length; ++i) {
            if (!(i in refs)) {
                continue;
            }
            if (!(refs[i] in locs)) {
                console.error(
                    `Assembler: label '${refs[i]}' used at ${i} not
defined`,
                );
                continue;
            }
            program[i] = locs[refs[i]];
        }
        return { program, locs };
    }

    public printProgram() {
        let ip = 0;
        for (const line of this.lines) {
            for (const label of line.labels ?? []) {
                console.log(`        ${label}:`);
            }
            const op = opToString(line.ins[0] as number)
```

```
                .padEnd(13, " ");
            const args = (line.ins.slice(1) as Lit[]).map((lit) =>
{
                if (typeof lit === "number") {
                    return lit;
                } else if (typeof lit === "boolean") {
                    return lit.toString();
                } else if (typeof lit === "string") {
                    return '"' +
                        lit.replaceAll("\\", "\\\
\").replaceAll("\0", "\\0")
                            .replaceAll("\n", "\
\n").replaceAll("\t", "\\t")
                            .replaceAll("\r", "\\r") +
                        '"';
                } else {
                    return lit.label;
                }
            }).join(", ");
            console.log(`${ip.toString().padStart(8, " ")}:
${op} ${args}`);
            ip += line.ins.map((lit) =>
                typeof lit === "string" ? lit.length + 1 : 1
            ).reduce((acc, curr) => acc + curr, 0);
        }
    }
}
```

compiler/main.ts

```
import { Compiler } from "./compiler.ts";

const { program } = await new Compiler(Deno.args[0]).compile();

await Deno.writeTextFile("out.slgbc", JSON.stringify(program));
```

compiler/lowerer.ts

```
import { Builtins, Ops } from "./arch.ts";
import { Expr, Stmt } from "./ast.ts";
import { LocalLeaf, Locals, LocalsFnRoot } from "./
lowerer_locals.ts";
import { Assembler, Label } from "./assembler.ts";
import { vtypeToString } from "./vtype.ts";
import { Pos } from "./token.ts";

export type FnNamesMap = { [pc: number]: string };

export class Lowerer {
    private program = Assembler.newRoot();
    private locals: Locals = new LocalsFnRoot();
    private fnStmtIdLabelMap: { [stmtId: number]: string } = {};
    private fnLabelNameMap: { [name: string]: string } = {};
```

```
private returnStack: Label[] = [];
private breakStack: Label[] = [];

public constructor(private lastPos: Pos) {}

public lower(stmts: Stmt[]) {
    this.addClearingSourceMap();
    this.program.add(Ops.PushPtr, { label: "main" });
    this.program.add(Ops.Call, 0);
    this.program.add(Ops.PushPtr, { label: "_exit" });
    this.program.add(Ops.Jump);
    this.scoutFnHeaders(stmts);
    for (const stmt of stmts) {
        this.lowerStaticStmt(stmt);
    }
    this.program.setLabel({ label: "_exit" });
    this.addSourceMap(this.lastPos);
    this.program.add(Ops.Pop);
}

public finish(): { program: number[]; fnNames: FnNamesMap } {
    const { program, locs } = this.program.assemble();
    const fnNames: FnNamesMap = {};
    for (const label in locs) {
        if (label in this.fnLabelNameMap) {
            fnNames[locs[label]] = this.fnLabelNameMap[label];
        }
    }
    return { program, fnNames };
}

private addSourceMap({ index, line, col }: Pos) {
    this.program.add(Ops.SourceMap, index, line, col);
}

private addClearingSourceMap() {
    this.program.add(Ops.SourceMap, 0, 1, 1);
}

private scoutFnHeaders(stmts: Stmt[]) {
    for (const stmt of stmts) {
        if (stmt.kind.type !== "fn") {
            continue;
        }
        const label = stmt.kind.ident === "main"
            ? "main"
            : `${stmt.kind.ident}_${stmt.id}`;
        this.fnStmtIdLabelMap[stmt.id] = label;
    }
}

private lowerStaticStmt(stmt: Stmt) {
    switch (stmt.kind.type) {
```

```
            case "fn":
                return this.lowerFnStmt(stmt);
            case "error":
            case "break":
            case "return":
            case "let":
            case "assign":
            case "expr":
        }
        throw new Error(`unhandled static statement
'${stmt.kind.type}'`);
    }

    private lowerStmt(stmt: Stmt) {
        switch (stmt.kind.type) {
            case "error":
                break;
            case "break":
                return this.lowerBreakStmt(stmt);
            case "return":
                return this.lowerReturnStmt(stmt);
            case "fn":
                return this.lowerFnStmt(stmt);
            case "let":
                return this.lowerLetStmt(stmt);
            case "assign":
                return this.lowerAssignStmt(stmt);
            case "expr":
                this.lowerExpr(stmt.kind.expr);
                this.program.add(Ops.Pop);
                return;
        }
        throw new Error(`unhandled stmt '${stmt.kind.type}'`);
    }

    private lowerAssignStmt(stmt: Stmt) {
        if (stmt.kind.type !== "assign") {
            throw new Error();
        }
        this.lowerExpr(stmt.kind.value);
        switch (stmt.kind.subject.kind.type) {
            case "field": {
                this.lowerExpr(stmt.kind.subject.kind.subject);
                this.program.add(Ops.PushString,
stmt.kind.subject.kind.value);
                this.program.add(Ops.Builtin, Builtins.StructSet);
                return;
            }
            case "index": {
                this.lowerExpr(stmt.kind.subject.kind.subject);
                this.lowerExpr(stmt.kind.subject.kind.value);
                this.program.add(Ops.Builtin, Builtins.ArraySet);
                return;
```

```
            }
            case "sym": {
                this.program.add(
                    Ops.StoreLocal,

this.locals.symId(stmt.kind.subject.kind.sym.ident),
                );
                return;
            }
            default:
                throw new Error();
        }
    }

    private lowerReturnStmt(stmt: Stmt) {
        if (stmt.kind.type !== "return") {
            throw new Error();
        }
        if (stmt.kind.expr) {
            this.lowerExpr(stmt.kind.expr);
        }
        this.addClearingSourceMap();
        this.program.add(Ops.PushPtr, this.returnStack.at(-1)!);
        this.program.add(Ops.Jump);
    }

    private lowerBreakStmt(stmt: Stmt) {
        if (stmt.kind.type !== "break") {
            throw new Error();
        }
        if (stmt.kind.expr) {
            this.lowerExpr(stmt.kind.expr);
        }
        this.addClearingSourceMap();
        this.program.add(Ops.PushPtr, this.breakStack.at(-1)!);
        this.program.add(Ops.Jump);
    }

    private lowerFnStmt(stmt: Stmt) {
        if (stmt.kind.type !== "fn") {
            throw new Error();
        }
        const label = stmt.kind.ident === "main"
            ? "main"
            : `${stmt.kind.ident}_${stmt.id}`;
        this.program.setLabel({ label });
        this.fnLabelNameMap[label] = stmt.kind.ident;
        this.addSourceMap(stmt.pos);

        const outerLocals = this.locals;
        const fnRoot = new LocalsFnRoot(outerLocals);
        const outerProgram = this.program;
```

```
        const returnLabel = this.program.makeLabel();
        this.returnStack.push(returnLabel);

        this.program = outerProgram.fork();
        this.locals = fnRoot;
        for (const { ident } of stmt.kind.params) {
            this.locals.allocSym(ident);
        }
        if (stmt.kind.anno?.ident === "builtin") {
            this.lowerFnBuiltinBody(stmt.kind.anno.values);
        } else if (stmt.kind.anno?.ident === "remainder") {
            this.program.add(Ops.Remainder);
        } else {
            this.lowerExpr(stmt.kind.body);
        }
        this.locals = outerLocals;

        const localAmount = fnRoot.stackReserved() -
            stmt.kind.params.length;
        for (let i = 0; i < localAmount; ++i) {
            outerProgram.add(Ops.PushNull);
        }

        this.returnStack.pop();
        this.program.setLabel(returnLabel);
        this.program.add(Ops.Return);

        outerProgram.join(this.program);
        this.program = outerProgram;
    }

    private lowerFnBuiltinBody(annoArgs: Expr[]) {
        if (annoArgs.length !== 1) {
            throw new Error("invalid # of arguments to builtin
annotation");
        }
        const anno = annoArgs[0];
        if (anno.kind.type !== "ident") {
            throw new Error(
                `unexpected argument type '${anno.kind.type}'
expected 'ident'`,
            );
        }
        const value = anno.kind.value;
        const builtin = Object.entries(Builtins).find((entry) =>
            entry[0] === value
        )?.[1];
        if (builtin === undefined) {
            throw new Error(
                `unrecognized builtin '${value}'`,
            );
        }
        this.program.add(Ops.Builtin, builtin);
```

```
    }

    private lowerLetStmt(stmt: Stmt) {
        if (stmt.kind.type !== "let") {
            throw new Error();
        }
        this.lowerExpr(stmt.kind.value);
        this.locals.allocSym(stmt.kind.param.ident);
        this.program.add(
            Ops.StoreLocal,
            this.locals.symId(stmt.kind.param.ident),
        );
    }

    private lowerExpr(expr: Expr) {
        switch (expr.kind.type) {
            case "error":
                break;
            case "sym":
                return this.lowerSymExpr(expr);
            case "null":
                break;
            case "int":
                return this.lowerIntExpr(expr);
            case "bool":
                return this.lowerBoolExpr(expr);
            case "string":
                return this.lowerStringExpr(expr);
            case "ident":
                break;
            case "group":
                return void this.lowerExpr(expr.kind.expr);
            case "field":
                break;
            case "index":
                return this.lowerIndexExpr(expr);
            case "call":
                return this.lowerCallExpr(expr);
            case "unary":
                return this.lowerUnaryExpr(expr);
            case "binary":
                return this.lowerBinaryExpr(expr);
            case "if":
                return this.lowerIfExpr(expr);
            case "loop":
                return this.lowerLoopExpr(expr);
            case "block":
                return this.lowerBlockExpr(expr);
        }
        throw new Error(`unhandled expr '${expr.kind.type}'`);
    }

    private lowerIndexExpr(expr: Expr) {
```

```
        if (expr.kind.type !== "index") {
            throw new Error();
        }
        this.lowerExpr(expr.kind.subject);
        this.lowerExpr(expr.kind.value);

        if (expr.kind.subject.vtype?.type == "array") {
            this.program.add(Ops.Builtin, Builtins.ArrayAt);
            return;
        }
        if (expr.kind.subject.vtype?.type == "string") {
            this.program.add(Ops.Builtin, Builtins.StringCharAt);
            return;
        }
        throw new Error(`unhandled index subject type
'${expr.kind.subject}'`);
    }

    private lowerSymExpr(expr: Expr) {
        if (expr.kind.type !== "sym") {
            throw new Error();
        }
        if (expr.kind.sym.type === "let") {
            const symId = this.locals.symId(expr.kind.ident);
            this.program.add(Ops.LoadLocal, symId);
            return;
        }
        if (expr.kind.sym.type === "fn_param") {
            this.program.add(
                Ops.LoadLocal,
                this.locals.symId(expr.kind.ident),
            );
            return;
        }
        if (expr.kind.sym.type === "fn") {
            const label =
this.fnStmtIdLabelMap[expr.kind.sym.stmt.id];
            this.program.add(Ops.PushPtr, { label });
            return;
        }
        throw new Error(`unhandled sym type
'${expr.kind.sym.type}'`);
    }

    private lowerIntExpr(expr: Expr) {
        if (expr.kind.type !== "int") {
            throw new Error();
        }
        this.program.add(Ops.PushInt, expr.kind.value);
    }

    private lowerBoolExpr(expr: Expr) {
        if (expr.kind.type !== "bool") {
```

```
            throw new Error();
        }
        this.program.add(Ops.PushBool, expr.kind.value);
    }

    private lowerStringExpr(expr: Expr) {
        if (expr.kind.type !== "string") {
            throw new Error();
        }
        this.program.add(Ops.PushString, expr.kind.value);
    }

    private lowerUnaryExpr(expr: Expr) {
        if (expr.kind.type !== "unary") {
            throw new Error();
        }
        this.lowerExpr(expr.kind.subject);
        const vtype = expr.kind.subject.vtype!;
        if (vtype.type === "bool") {
            switch (expr.kind.unaryType) {
                case "not":
                    this.program.add(Ops.Not);
                    return;
                default:
            }
        }
        if (vtype.type === "int") {
            switch (expr.kind.unaryType) {
                case "-": {
                    this.program.add(Ops.PushInt, 0);
                    this.program.add(Ops.Swap);
                    this.program.add(Ops.Subtract);
                    return;
                }
                default:
            }
        }
        throw new Error(
            `unhandled unary` +
                ` '${vtypeToString(expr.vtype!)}' aka. ` +
                ` ${expr.kind.unaryType}` +
                ` '${vtypeToString(expr.kind.subject.vtype!)}'`,
        );
    }

    private lowerBinaryExpr(expr: Expr) {
        if (expr.kind.type !== "binary") {
            throw new Error();
        }
        const vtype = expr.kind.left.vtype!;
        if (vtype.type === "bool") {
            if (["or", "and"].includes(expr.kind.binaryType)) {
                const shortCircuitLabel = this.program.makeLabel();
```

```
        this.lowerExpr(expr.kind.left);
        this.program.add(Ops.Duplicate);
        if (expr.kind.binaryType === "and") {
            this.program.add(Ops.Not);
        }
        this.program.add(Ops.PushPtr, shortCircuitLabel);
        this.program.add(Ops.JumpIfTrue);
        this.program.add(Ops.Pop);
        this.lowerExpr(expr.kind.right);
        this.program.setLabel(shortCircuitLabel);
        return;
    }
}
this.lowerExpr(expr.kind.left);
this.lowerExpr(expr.kind.right);
if (vtype.type === "int") {
    switch (expr.kind.binaryType) {
        case "+":
            this.program.add(Ops.Add);
            return;
        case "-":
            this.program.add(Ops.Subtract);
            return;
        case "*":
            this.program.add(Ops.Multiply);
            return;
        case "/":
            this.program.add(Ops.Multiply);
            return;
        case "==":
            this.program.add(Ops.Equal);
            return;
        case "!=":
            this.program.add(Ops.Equal);
            this.program.add(Ops.Not);
            return;
        case "<":
            this.program.add(Ops.LessThan);
            return;
        case ">":
            this.program.add(Ops.Swap);
            this.program.add(Ops.LessThan);
            return;
        case "<=":
            this.program.add(Ops.Swap);
            this.program.add(Ops.LessThan);
            this.program.add(Ops.Not);
            return;
        case ">=":
            this.program.add(Ops.LessThan);
            this.program.add(Ops.Not);
            return;
        default:
```

```
            }
        }
        if (vtype.type === "string") {
            if (expr.kind.binaryType === "+") {
                this.program.add(Ops.Builtin,
Builtins.StringConcat);
                return;
            }
            if (expr.kind.binaryType === "==") {
                this.program.add(Ops.Builtin,
Builtins.StringEqual);
                return;
            }
            if (expr.kind.binaryType === "!=") {
                this.program.add(Ops.Builtin,
Builtins.StringEqual);
                this.program.add(Ops.Not);
                return;
            }
        }
        throw new Error(
            `unhandled binaryType` +
                ` '${vtypeToString(expr.vtype!)}' aka. ` +
                ` '${vtypeToString(expr.kind.left.vtype!)}'` +
                ` ${expr.kind.binaryType}` +
                ` '${vtypeToString(expr.kind.left.vtype!)}'`,
        );
    }

    private lowerCallExpr(expr: Expr) {
        if (expr.kind.type !== "call") {
            throw new Error();
        }
        for (const arg of expr.kind.args) {
            this.lowerExpr(arg);
        }
        this.lowerExpr(expr.kind.subject);
        this.program.add(Ops.Call, expr.kind.args.length);
    }

    private lowerIfExpr(expr: Expr) {
        if (expr.kind.type !== "if") {
            throw new Error();
        }

        const falseLabel = this.program.makeLabel();
        const doneLabel = this.program.makeLabel();

        this.lowerExpr(expr.kind.cond);

        this.program.add(Ops.Not);
        this.addClearingSourceMap();
        this.program.add(Ops.PushPtr, falseLabel);
```

```
        this.program.add(Ops.JumpIfTrue);

        this.addSourceMap(expr.kind.truthy.pos);
        this.lowerExpr(expr.kind.truthy);

        this.addClearingSourceMap();
        this.program.add(Ops.PushPtr, doneLabel);
        this.program.add(Ops.Jump);

        this.program.setLabel(falseLabel);

        if (expr.kind.falsy) {
            this.addSourceMap(expr.kind.elsePos!);
            this.lowerExpr(expr.kind.falsy);
        } else {
            this.program.add(Ops.PushNull);
        }

        this.program.setLabel(doneLabel);
    }

    private lowerLoopExpr(expr: Expr) {
        if (expr.kind.type !== "loop") {
            throw new Error();
        }
        const continueLabel = this.program.makeLabel();
        const breakLabel = this.program.makeLabel();

        this.breakStack.push(breakLabel);

        this.program.setLabel(continueLabel);
        this.addSourceMap(expr.kind.body.pos);
        this.lowerExpr(expr.kind.body);
        this.program.add(Ops.Pop);
        this.addClearingSourceMap();
        this.program.add(Ops.PushPtr, continueLabel);
        this.program.add(Ops.Jump);
        this.program.setLabel(breakLabel);
        if (expr.vtype!.type === "null") {
            this.program.add(Ops.PushNull);
        }
        this.breakStack.pop();
    }

    private lowerBlockExpr(expr: Expr) {
        if (expr.kind.type !== "block") {
            throw new Error();
        }
        const outerLocals = this.locals;
        this.locals = new LocalLeaf(this.locals);
        this.scoutFnHeaders(expr.kind.stmts);
        for (const stmt of expr.kind.stmts) {
            this.addSourceMap(stmt.pos);
```

```
            this.lowerStmt(stmt);
        }
        if (expr.kind.expr) {
            this.addSourceMap(expr.kind.expr.pos);
            this.lowerExpr(expr.kind.expr);
        } else {
            this.program.add(Ops.PushNull);
        }
        this.locals = outerLocals;
    }

    public printProgram() {
        this.program.printProgram();
    }
}
```

compiler/compiler.ts

```
import { AstCreator } from "./ast.ts";
import { Checker } from "./checker.ts";
import { CompoundAssignDesugarer } from "./desugar/
compound_assign.ts";
import { SpecialLoopDesugarer } from "./desugar/special_loop.ts";
import { Reporter } from "./info.ts";
import { Lexer } from "./lexer.ts";
import { FnNamesMap, Lowerer } from "./lowerer.ts";
import { Parser } from "./parser.ts";
import { Resolver } from "./resolver.ts";

export type CompiledFile = {
    filepath: string;
    program: number[];
};

export type CompileResult = {
    program: number[];
    fnNames: FnNamesMap;
};

export class Compiler {
    private astCreator = new AstCreator();
    private reporter = new Reporter();

    public constructor(private startFilePath: string) {}

    public async compile(): Promise<CompileResult> {
        const text = await Deno.readTextFile(this.startFilePath);

        const lexer = new Lexer(text, this.reporter);

        const parser = new Parser(lexer, this.astCreator,
this.reporter);
        const ast = parser.parse();
```

```
        new SpecialLoopDesugarer(this.astCreator).desugar(ast);

        new Resolver(this.reporter).resolve(ast);

        new CompoundAssignDesugarer(this.astCreator).desugar(ast);

        new Checker(this.reporter).check(ast);

        if (this.reporter.errorOccured()) {
            console.error("Errors occurred, stopping
compilation.");
            Deno.exit(1);
        }

        const lowerer = new Lowerer(lexer.currentPos());
        lowerer.lower(ast);
        // lowerer.printProgram();
        const { program, fnNames } = lowerer.finish();

        return { program, fnNames };
    }
}
```

compiler/ast.ts

```
import { Pos } from "./token.ts";
import { VType } from "./vtype.ts";

export type Stmt = {
    kind: StmtKind;
    pos: Pos;
    id: number;
};

export type StmtKind =
    | { type: "error" }
    | { type: "import"; path: Expr }
    | { type: "break"; expr?: Expr }
    | { type: "return"; expr?: Expr }
    | {
        type: "fn";
        ident: string;
        params: Param[];
        returnType?: EType;
        body: Expr;
        anno?: Anno;
        vtype?: VType;
    }
    | { type: "let"; param: Param; value: Expr }
    | { type: "assign"; assignType: AssignType; subject: Expr;
value: Expr }
    | { type: "expr"; expr: Expr };
```

```
export type AssignType = "=" | "+=" | "-=";

export type Expr = {
    kind: ExprKind;
    pos: Pos;
    vtype?: VType;
    id: number;
};

export type ExprKind =
    | { type: "error" }
    | { type: "int"; value: number }
    | { type: "string"; value: string }
    | { type: "ident"; value: string }
    | { type: "group"; expr: Expr }
    | { type: "field"; subject: Expr; value: string }
    | { type: "index"; subject: Expr; value: Expr }
    | { type: "call"; subject: Expr; args: Expr[] }
    | { type: "unary"; unaryType: UnaryType; subject: Expr }
    | { type: "binary"; binaryType: BinaryType; left: Expr; right:
Expr }
    | { type: "if"; cond: Expr; truthy: Expr; falsy?: Expr;
elsePos?: Pos }
    | { type: "bool"; value: boolean }
    | { type: "null" }
    | { type: "loop"; body: Expr }
    | { type: "block"; stmts: Stmt[]; expr?: Expr }
    | {
        type: "sym";
        ident: string;
        sym: Sym;
    }
    | { type: "while"; cond: Expr; body: Expr }
    | { type: "for_in"; param: Param; value: Expr; body: Expr }
    | {
        type: "for";
        decl?: Stmt;
        cond?: Expr;
        incr?: Stmt;
        body: Expr;
    };

export type UnaryType = "not" | "-";
export type BinaryType =
    | "+"
    | "*"
    | "=="
    | "-"
    | "/"
    | "!="
    | "<"
    | ">"
```

```
    | "<="
    | ">="
    | "or"
    | "and";

export type Param = {
    ident: string;
    etype?: EType;
    pos: Pos;
    vtype?: VType;
};

export type Sym = {
    ident: string;
    pos?: Pos;
} & SymKind;

export type SymKind =
    | { type: "let"; stmt: Stmt; param: Param }
    | { type: "let_static"; stmt: Stmt; param: Param }
    | { type: "fn"; stmt: Stmt }
    | { type: "fn_param"; param: Param }
    | { type: "closure"; inner: Sym }
    | { type: "builtin"; builtinId: number };

export type EType = {
    kind: ETypeKind;
    pos: Pos;
    id: number;
};

export type ETypeKind =
    | { type: "error" }
    | { type: "ident"; value: string }
    | { type: "array"; inner: EType }
    | { type: "struct"; fields: Param[] };

export type ETypeParam = {
    ident: string;
    pos: Pos;
    vtype?: VType;
};

export type Anno = {
    ident: string;
    values: Expr[];
    pos: Pos;
};

export class AstCreator {
    private nextNodeId = 0;

    public stmt(kind: StmtKind, pos: Pos): Stmt {
```

```
        const id = this.nextNodeId;
        this.nextNodeId += 1;
        return { kind, pos, id };
    }

    public expr(kind: ExprKind, pos: Pos): Expr {
        const id = this.nextNodeId;
        this.nextNodeId += 1;
        return { kind, pos, id };
    }

    public etype(kind: ETypeKind, pos: Pos): EType {
        const id = this.nextNodeId;
        this.nextNodeId += 1;
        return { kind, pos, id };
    }
}
```

compiler/lowerer_locals.ts

```
export interface Locals {
    reserveAmount(id: number): void;
    allocSym(ident: string): void;
    symId(ident: string): number;

    currentLocalIdCounter(): number;
}

export class LocalsFnRoot implements Locals {
    private localsAmount = 0;
    private symLocalMap: { [key: string]: number } = {};
    private localIdCounter: number;

    constructor(private parent?: Locals) {
        this.localIdCounter = parent?.currentLocalIdCounter() ?? 0;
    }

    public reserveAmount(amount: number): void {
        this.localsAmount = Math.max(amount, this.localsAmount);
    }

    public allocSym(ident: string) {
        this.symLocalMap[ident] = this.localIdCounter;
        this.localIdCounter++;
        this.reserveAmount(this.localIdCounter);
    }

    public symId(ident: string): number {
        if (ident in this.symLocalMap) {
            return this.symLocalMap[ident];
        }
        if (this.parent) {
            return this.parent.symId(ident);
```

```
        }
        throw new Error(`undefined symbol '${ident}'`);
    }

    public stackReserved(): number {
        return this.localsAmount;
    }

    public currentLocalIdCounter(): number {
        return this.localIdCounter;
    }
}

export class LocalLeaf implements Locals {
    private symLocalMap: { [key: string]: number } = {};
    private localIdCounter: number;

    constructor(private parent: Locals) {
        this.localIdCounter = parent.currentLocalIdCounter();
    }

    public reserveAmount(amount: number): void {
        this.parent.reserveAmount(amount);
    }

    public allocSym(ident: string) {
        this.symLocalMap[ident] = this.localIdCounter;
        this.localIdCounter++;
        this.reserveAmount(this.localIdCounter);
    }

    public symId(ident: string): number {
        if (ident in this.symLocalMap) {
            return this.symLocalMap[ident];
        }
        return this.parent.symId(ident);
    }

    public currentLocalIdCounter(): number {
        return this.localIdCounter;
    }
}
```

compiler/parser.ts

```
import {
    Anno,
    AssignType,
    AstCreator,
    BinaryType,
    EType,
    ETypeKind,
    Expr,
```

```
    ExprKind,
    Param,
    Stmt,
    StmtKind,
    UnaryType,
} from "./ast.ts";
import { printStackTrace, Reporter } from "./info.ts";
import { Lexer } from "./lexer.ts";
import { Pos, Token } from "./token.ts";

export class Parser {
    private currentToken: Token | null;

    public constructor(
        private lexer: Lexer,
        private astCreator: AstCreator,
        private reporter: Reporter,
    ) {
        this.currentToken = lexer.next();
    }

    public parse(): Stmt[] {
        return this.parseStmts();
    }

    private parseStmts(): Stmt[] {
        const stmts: Stmt[] = [];
        while (!this.done()) {
            if (this.test("fn")) {
                stmts.push(this.parseFn());
            } else if (
                this.test("let") || this.test("return") ||
this.test("break")
            ) {
                stmts.push(this.parseSingleLineBlockStmt());
                this.eatSemicolon();
            } else if (
                ["{", "if", "loop", "while", "for"].some((tt) =>
this.test(tt))
            ) {
                const expr = this.parseMultiLineBlockExpr();
                stmts.push(this.stmt({ type: "expr", expr },
expr.pos));
            } else {
                stmts.push(this.parseAssign());
                this.eatSemicolon();
            }
        }
        return stmts;
    }

    private parseMultiLineBlockExpr(): Expr {
        const pos = this.pos();
```

```
        if (this.test("{")) {
            return this.parseBlock();
        }
        if (this.test("if")) {
            return this.parseIf();
        }
        if (this.test("loop")) {
            return this.parseLoop();
        }
        if (this.test("while")) {
            return this.parseWhile();
        }
        if (this.test("for")) {
            return this.parseFor();
        }
        this.report("expected expr");
        return this.expr({ type: "error" }, pos);
    }

    private parseSingleLineBlockStmt(): Stmt {
        const pos = this.pos();
        if (this.test("let")) {
            return this.parseLet();
        }
        if (this.test("return")) {
            return this.parseReturn();
        }
        if (this.test("break")) {
            return this.parseBreak();
        }
        this.report("expected stmt");
        return this.stmt({ type: "error" }, pos);
    }

    private eatSemicolon() {
        if (!this.test(";")) {
            this.report(
                `expected ';', got '${this.currentToken?.type ??
"eof"}'`,
            );
            return;
        }
        this.step();
    }

    private parseExpr(): Expr {
        return this.parseBinary();
    }

    private parseBlock(): Expr {
        const pos = this.pos();
        this.step();
        let stmts: Stmt[] = [];
```

```
        while (!this.done()) {
            if (this.test("}")) {
                this.step();
                return this.expr({ type: "block", stmts }, pos);
            } else if (
                this.test("return") || this.test("break") ||
this.test("let")
            ) {
                stmts.push(this.parseSingleLineBlockStmt());
                this.eatSemicolon();
            } else if (this.test("fn")) {
                stmts.push(this.parseSingleLineBlockStmt());
                stmts.push(this.parseFn());
            } else if (
                ["{", "if", "loop", "while", "for"].some((tt) =>
this.test(tt))
            ) {
                const expr = this.parseMultiLineBlockExpr();
                if (this.test("}")) {
                    this.step();
                    return this.expr({ type: "block", stmts, expr
}, expr.pos);
                }
                stmts.push(this.stmt({ type: "expr", expr },
expr.pos));
            } else {
                const expr = this.parseExpr();
                if (this.test("=") || this.test("+=") ||
this.test("-=")) {
                    const assignType = this.current().type as
AssignType;
                    this.step();
                    const value = this.parseExpr();
                    this.eatSemicolon();
                    stmts.push(
                        this.stmt(
                            {
                                type: "assign",
                                assignType,
                                subject: expr,
                                value,
                            },
                            expr.pos,
                        ),
                    );
                } else if (this.test(";")) {
                    this.step();
                    stmts.push(this.stmt({ type: "expr", expr },
expr.pos));
                } else if (this.test("}")) {
                    this.step();
                    return this.expr({ type: "block", stmts, expr
}, pos);
```

```
                } else {
                    this.report("expected ';' or '}'");
                    return this.expr({ type: "error" },
this.pos());
                }
            }
        }
        this.report("expected '}'");
        return this.expr({ type: "error" }, pos);
    }

    private parseFn(): Stmt {
        const pos = this.pos();
        this.step();
        if (!this.test("ident")) {
            this.report("expected ident");
            return this.stmt({ type: "error" }, pos);
        }
        const ident = this.current().identValue!;
        this.step();
        if (!this.test("(")) {
            this.report("expected '('");
            return this.stmt({ type: "error" }, pos);
        }
        const params = this.parseFnParams();
        let returnType: EType | null = null;
        if (this.test("->")) {
            this.step();
            returnType = this.parseEType();
        }

        let anno: Anno | null = null;
        if (this.test("#")) {
            anno = this.parseAnno();
        }
        if (!this.test("{")) {
            this.report("expected block");
            return this.stmt({ type: "error" }, pos);
        }
        const body = this.parseBlock();
        return this.stmt(
            {
                type: "fn",
                ident,
                params,
                returnType: returnType !== null ? returnType :
undefined,
                body,
                anno: anno != null ? anno : undefined,
            },
            pos,
        );
    }
```

```
private parseAnnoArgs(): Expr[] {
    this.step();
    if (!this.test("(")) {
        this.report("expected '('");
        return [];
    }
    this.step();
    const annoArgs: Expr[] = [];
    if (!this.test(")")) {
        annoArgs.push(this.parseExpr());
        while (this.test(",")) {
            this.step();
            if (this.test(")")) {
                break;
            }
            annoArgs.push(this.parseExpr());
        }
    }
    if (!this.test(")")) {
        this.report("expected ')'");
        return [];
    }
    this.step();
    return annoArgs;
}

private parseAnno(): Anno | null {
    const pos = this.pos();
    this.step();
    if (!this.test("[")) {
        this.report("expected '['");
        return null;
    }
    this.step();
    if (!this.test("ident")) {
        this.report("expected identifier");
        return null;
    }
    const ident = this.current().identValue!;
    const values = this.parseAnnoArgs();
    if (!this.test("]")) {
        this.report("expected ']'");
        return null;
    }
    this.step();
    return { ident, pos, values };
}

private parseFnParams(): Param[] {
    this.step();
    if (this.test(")")) {
        this.step();
```

```
            return [];
        }
        const params: Param[] = [];
        const paramResult = this.parseParam();
        if (!paramResult.ok) {
            return [];
        }
        params.push(paramResult.value);
        while (this.test(",")) {
            this.step();
            if (this.test(")")) {
                break;
            }
            const paramResult = this.parseParam();
            if (!paramResult.ok) {
                return [];
            }
            params.push(paramResult.value);
        }
        if (!this.test(")")) {
            this.report("expected ')'");
            return params;
        }
        this.step();
        return params;
    }

    private parseParam(): { ok: true; value: Param } | { ok: false
} {
        const pos = this.pos();
        if (this.test("ident")) {
            const ident = this.current().identValue!;
            this.step();
            if (this.test(":")) {
                this.step();
                const etype = this.parseEType();
                return { ok: true, value: { ident, etype, pos } };
            }
            return { ok: true, value: { ident, pos } };
        }
        this.report("expected param");
        return { ok: false };
    }

    private parseLet(): Stmt {
        const pos = this.pos();
        this.step();
        const paramResult = this.parseParam();
        if (!paramResult.ok) {
            return this.stmt({ type: "error" }, pos);
        }
        const param = paramResult.value;
        if (!this.test("=")) {
```

```
            this.report("expected '='");
            return this.stmt({ type: "error" }, pos);
        }
        this.step();
        const value = this.parseExpr();
        return this.stmt({ type: "let", param, value }, pos);
    }

    private parseAssign(): Stmt {
        const pos = this.pos();
        const subject = this.parseExpr();
        if (this.test("=") || this.test("+=") || this.test("-=")) {
            const assignType = this.current().type as AssignType;
            this.step();
            const value = this.parseExpr();
            return this.stmt({
                type: "assign",
                assignType,
                subject,
                value,
            }, pos);
        }
        return this.stmt({ type: "expr", expr: subject }, pos);
    }

    private parseReturn(): Stmt {
        const pos = this.pos();
        this.step();
        if (this.test(";")) {
            return this.stmt({ type: "return" }, pos);
        }
        const expr = this.parseExpr();
        return this.stmt({ type: "return", expr }, pos);
    }

    private parseBreak(): Stmt {
        const pos = this.pos();
        this.step();
        if (this.test(";")) {
            return this.stmt({ type: "break" }, pos);
        }
        const expr = this.parseExpr();
        return this.stmt({ type: "break", expr }, pos);
    }

    private parseLoop(): Expr {
        const pos = this.pos();
        this.step();
        if (!this.test("{")) {
            this.report("expected '{'");
            return this.expr({ type: "error" }, pos);
        }
        const body = this.parseExpr();
```

```
        return this.expr({ type: "loop", body }, pos);
    }

    private parseWhile(): Expr {
        const pos = this.pos();
        this.step();
        const cond = this.parseExpr();
        if (!this.test("{")) {
            this.report("expected '{'");
            return this.expr({ type: "error" }, pos);
        }
        const body = this.parseExpr();
        return this.expr({ type: "while", cond, body }, pos);
    }

    private parseFor(): Expr {
        const pos = this.pos();
        this.step();

        if (this.test("(")) {
            return this.parseForClassicTail(pos);
        }

        const paramRes = this.parseParam();
        if (!paramRes.ok) {
            return this.expr({ type: "error" }, pos);
        }
        const param = paramRes.value;

        if (!this.test("in")) {
            this.report("expected 'in'");
            return this.expr({ type: "error" }, pos);
        }
        this.step();
        const value = this.parseExpr();

        if (!this.test("{")) {
            this.report("expected '{'");
            return this.expr({ type: "error" }, pos);
        }
        const body = this.parseExpr();
        return this.expr({ type: "for_in", param, value, body },
pos);
    }

    private parseForClassicTail(pos: Pos): Expr {
        this.step();
        let decl: Stmt | undefined;
        if (!this.test(";")) {
            decl = this.parseLet();
        }
        if (!this.test(";")) {
            this.report("expected ';'");
```

```
            return this.expr({ type: "error" }, pos);
        }
        this.step();
        let cond: Expr | undefined;
        if (!this.test(";")) {
            cond = this.parseExpr();
        }
        if (!this.test(";")) {
            this.report("expected ';'");
            return this.expr({ type: "error" }, pos);
        }
        this.step();
        let incr: Stmt | undefined;
        if (!this.test(")")) {
            incr = this.parseAssign();
        }
        if (!this.test(")")) {
            this.report("expected '}'");
            return this.expr({ type: "error" }, pos);
        }
        this.step();

        if (!this.test("{")) {
            this.report("expected '{'");
            return this.expr({ type: "error" }, pos);
        }
        const body = this.parseExpr();
        return this.expr({ type: "for", decl, cond, incr, body },
pos);
    }

    private parseIf(): Expr {
        const pos = this.pos();
        this.step();
        const cond = this.parseExpr();
        if (!this.test("{")) {
            this.report("expected block");
            return this.expr({ type: "error" }, pos);
        }
        const truthy = this.parseBlock();
        if (!this.test("else")) {
            return this.expr({ type: "if", cond, truthy }, pos);
        }
        const elsePos = this.pos();
        this.step();
        if (this.test("if")) {
            const falsy = this.parseIf();
            return this.expr({ type: "if", cond, truthy, falsy,
elsePos }, pos);
        }
        if (!this.test("{")) {
            this.report("expected block");
            return this.expr({ type: "error" }, pos);
```

```
        }
        const falsy = this.parseBlock();
        return this.expr({ type: "if", cond, truthy, falsy, elsePos
}, pos);
    }

    private parseBinary(): Expr {
        return this.parseOr();
    }

    private parseOr(): Expr {
        const pos = this.pos();
        let left = this.parseAnd();
        while (true) {
            if (this.test("or")) {
                left = this.parBinTail(left, pos, this.parseAnd,
"or");
            } else {
                break;
            }
        }
        return left;
    }

    private parseAnd(): Expr {
        const pos = this.pos();
        let left = this.parseEquality();
        while (true) {
            if (this.test("and")) {
                left = this.parBinTail(left, pos,
this.parseEquality, "and");
            } else {
                break;
            }
        }
        return left;
    }

    private parseEquality(): Expr {
        const pos = this.pos();
        const left = this.parseComparison();
        if (this.test("==")) {
            return this.parBinTail(left, pos, this.parseComparison,
"==");
        }
        if (this.test("!=")) {
            return this.parBinTail(left, pos, this.parseComparison,
"!=");
        }
        return left;
    }

    private parseComparison(): Expr {
```

```
        const pos = this.pos();
        const left = this.parseAddSub();
        if (this.test("<")) {
            return this.parBinTail(left, pos, this.parseAddSub,
"<");
        }
        if (this.test(">")) {
            return this.parBinTail(left, pos, this.parseAddSub,
">");
        }
        if (this.test("<=")) {
            return this.parBinTail(left, pos, this.parseAddSub,
"<=");
        }
        if (this.test(">=")) {
            return this.parBinTail(left, pos, this.parseAddSub,
">=");
        }
        return left;
    }

    private parseAddSub(): Expr {
        const pos = this.pos();
        let left = this.parseMulDiv();
        while (true) {
            if (this.test("+")) {
                left = this.parBinTail(left, pos, this.parseMulDiv,
"+");
                continue;
            }
            if (this.test("-")) {
                left = this.parBinTail(left, pos, this.parseMulDiv,
"-");
                continue;
            }
            break;
        }
        return left;
    }

    private parseMulDiv(): Expr {
        const pos = this.pos();
        let left = this.parsePrefix();
        while (true) {
            if (this.test("*")) {
                left = this.parBinTail(left, pos, this.parsePrefix,
"*");
                continue;
            }
            if (this.test("/")) {
                left = this.parBinTail(left, pos, this.parsePrefix,
"/");
                continue;
```

```
            }
            break;
        }
        return left;
    }

    private parBinTail(
        left: Expr,
        pos: Pos,
        parseRight: (this: Parser) => Expr,
        binaryType: BinaryType,
    ): Expr {
        this.step();
        const right = parseRight.call(this);
        return this.expr(
            { type: "binary", binaryType, left, right },
            pos,
        );
    }

    private parsePrefix(): Expr {
        const pos = this.pos();
        if (this.test("not") || this.test("-")) {
            const unaryType = this.current().type as UnaryType;
            this.step();
            const subject = this.parsePrefix();
            return this.expr({ type: "unary", unaryType, subject },
pos);
        }
        return this.parsePostfix();
    }

    private parsePostfix(): Expr {
        let subject = this.parseOperand();
        while (true) {
            const pos = this.pos();
            if (this.test(".")) {
                this.step();
                if (!this.test("ident")) {
                    this.report("expected ident");
                    return this.expr({ type: "error" }, pos);
                }
                const value = this.current().identValue!;
                this.step();
                subject = this.expr({ type: "field", subject, value
}, pos);
                continue;
            }
            if (this.test("[")) {
                this.step();
                const value = this.parseExpr();
                if (!this.test("]")) {
                    this.report("expected ']'");
```

```
                    return this.expr({ type: "error" }, pos);
                }
                this.step();
                subject = this.expr({ type: "index", subject, value
}, pos);
                continue;
            }
            if (this.test("(")) {
                this.step();
                let args: Expr[] = [];
                if (!this.test(")")) {
                    args.push(this.parseExpr());
                    while (this.test(",")) {
                        this.step();
                        if (this.test(")")) {
                            break;
                        }
                        args.push(this.parseExpr());
                    }
                }
                if (!this.test(")")) {
                    this.report("expected ')'");
                    return this.expr({ type: "error" }, pos);
                }
                this.step();
                subject = this.expr({ type: "call", subject, args
}, pos);
                continue;
            }
            break;
        }
        return subject;
    }

    private parseOperand(): Expr {
        const pos = this.pos();
        if (this.test("ident")) {
            const value = this.current().identValue!;
            this.step();
            return this.expr({ type: "ident", value }, pos);
        }
        if (this.test("int")) {
            const value = this.current().intValue!;
            this.step();
            return this.expr({ type: "int", value }, pos);
        }
        if (this.test("string")) {
            const value = this.current().stringValue!;
            this.step();
            return this.expr({ type: "string", value }, pos);
        }
        if (this.test("false")) {
            this.step();
```

```
            return this.expr({ type: "bool", value: false }, pos);
        }
        if (this.test("true")) {
            this.step();
            return this.expr({ type: "bool", value: true }, pos);
        }
        if (this.test("null")) {
            this.step();
            return this.expr({ type: "null" }, pos);
        }
        if (this.test("(")) {
            this.step();
            const expr = this.parseExpr();
            if (!this.test(")")) {
                this.report("expected ')'");
                return this.expr({ type: "error" }, pos);
            }
            this.step();
            return this.expr({ type: "group", expr }, pos);
        }
        if (this.test("{")) {
            return this.parseBlock();
        }
        if (this.test("if")) {
            return this.parseIf();
        }
        if (this.test("loop")) {
            return this.parseLoop();
        }

        this.report("expected expr", pos);
        this.step();
        return this.expr({ type: "error" }, pos);
    }

    private parseEType(): EType {
        const pos = this.pos();
        if (this.test("ident")) {
            const ident = this.current().identValue!;
            this.step();
            return this.etype({ type: "ident", value: ident },
pos);
        }
        if (this.test("[")) {
            this.step();
            const inner = this.parseEType();
            if (!this.test("]")) {
                this.report("expected ']'", pos);
                return this.etype({ type: "error" }, pos);
            }
            this.step();
            return this.etype({ type: "array", inner }, pos);
        }
```

```
        if (this.test("struct")) {
            this.step();
            if (!this.test("{")) {
                this.report("expected '{'");
                return this.etype({ type: "error" }, pos);
            }
            const fields = this.parseETypeStructFields();
            return this.etype({ type: "struct", fields }, pos);
        }
        this.report("expected type");
        return this.etype({ type: "error" }, pos);
    }

    private parseETypeStructFields(): Param[] {
        this.step();
        if (this.test("}")) {
            this.step();
            return [];
        }
        const params: Param[] = [];
        const paramResult = this.parseParam();
        if (!paramResult.ok) {
            return [];
        }
        params.push(paramResult.value);
        while (this.test(",")) {
            this.step();
            if (this.test("}")) {
                break;
            }
            const paramResult = this.parseParam();
            if (!paramResult.ok) {
                return [];
            }
            params.push(paramResult.value);
        }
        if (!this.test("}")) {
            this.report("expected '}'");
            return params;
        }
        this.step();
        return params;
    }

    private step() {
        this.currentToken = this.lexer.next();
    }
    private done(): boolean {
        return this.currentToken == null;
    }
    private current(): Token {
        return this.currentToken!;
    }
```

```
    private pos(): Pos {
        if (this.done()) {
            return this.lexer.currentPos();
        }
        return this.current().pos;
    }

    private test(type: string): boolean {
        return !this.done() && this.current().type === type;
    }

    private report(msg: string, pos = this.pos()) {
        console.log(`Parser: ${msg} at ${pos.line}:${pos.col}`);
        this.reporter.reportError({
            msg,
            pos,
            reporter: "Parser",
        });
        printStackTrace();
    }

    private stmt(kind: StmtKind, pos: Pos): Stmt {
        return this.astCreator.stmt(kind, pos);
    }

    private expr(kind: ExprKind, pos: Pos): Expr {
        return this.astCreator.expr(kind, pos);
    }

    private etype(kind: ETypeKind, pos: Pos): EType {
        return this.astCreator.etype(kind, pos);
    }
}
```

compiler/lib.ts

```
import { Compiler, CompileResult } from "./compiler.ts";

export async function compileWithDebug(path: string):
Promise<CompileResult> {
    const { program, fnNames } = await new
Compiler(path).compile();
    return { program, fnNames };
}
```