Part II Project: Progress Report

Martin Walls

Achieved work

```
Source code
     |
     v
    AST
     |
     v
Three-address
   code IR   ⟲
     |
     v
  Relooper
  algorithm
     |
     v
 Wasm binary
     |
     v
NodeJS runtime
```
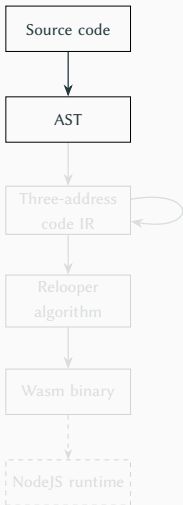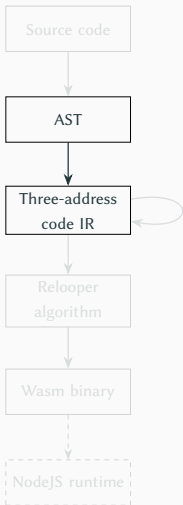
- Rust parser generator: LALRPOP
- Custom lexer, to handle `typedef` definitions (context-sensitivity).
- Avoiding ambiguities e.g. dangling `else`

Achieved work

- Defined three-address code representation
- For each node in the AST, defined transformation to three-address code
- Complexity:
    - Switch statement logic: fall-through and default cases
    - Assignment: evaluating an expression either as loading a value or storing to that address
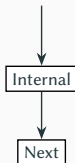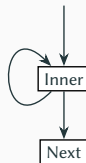
Achieved work

Source code

AST

Three-address
code IR

Relooper
algorithm

Wasm binary

NodeJS runtime
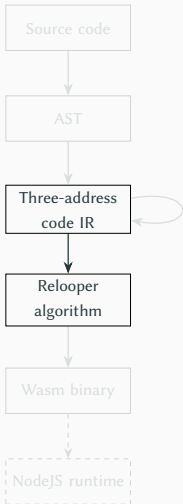
Implemented the Relooper algorithm

- Turning the linear sequence of IR
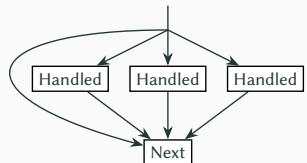 instructions into a structure of 'blocks'

Simple

Internal

Next

Loop
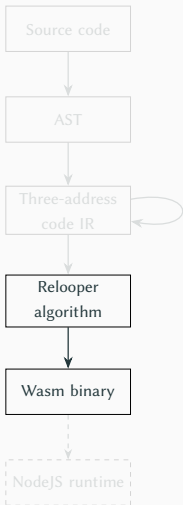
Inner

Next

Multiple

Handled   Handled   Handled

Next

```
Source code
   |
   v
  AST
   |
   v
Three-address
  code IR
   |
   v
 Relooper
 algorithm
   |
   v
Wasm binary
   |
   v
NodeJS runtime
```
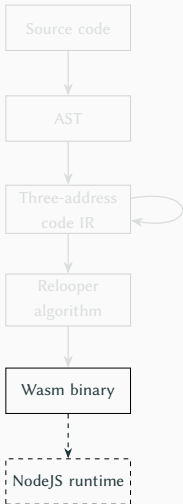
Target code generation

- Defined WebAssembly instructions for each IR instruction
- Pushing/popping function call stack frames
- Updating stack and frame pointers
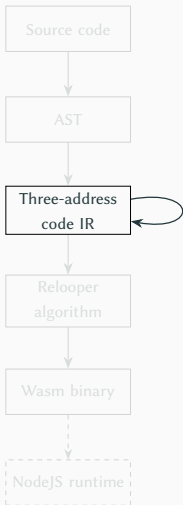- Allocating addresses for variables

Achieved work

NodeJS runtime:

- Instantiate WebAssembly module
- Initialise memory, and store program arguments
- Implemented some of the C standard library, e.g. `printf`

Implemented optimisations

Source code

AST

Three-address
code IR

Relooper
algorithm

Wasm binary

NodeJS runtime

- Tail-call optimisation
  - Find recursive tail-calls in each procedure
  - Instead, set the parameter variables to the new values and loop back to the entry point
- Unreachable procedure elimination
  - Generate call graph
  - Walk call graph, marking all reached functions
  - Remove all unmarked functions