6.110 Computer Language Engineering

Recitation 9: Phase 4 infosession

April 4, 2025

Weekly updates ←

Phase 4 info

Wrapping up phase 3…

- **Project phase 3 is due today 11:59PM!!!**
  - **The report is due tomorrow at 11:59PM**
  - Remember to add your teammates to the submission!
  - If you need last-minute help, please come to OH today from 2-7pm.

New releases

- Project phase 4 has been released, due **Friday, April 18**
- Miniquiz (will be posted soon) and Weekly Check-in are due **Thursday, April 10**
  - Reminder: these are graded on completion – please submit!!

Schedule... **Week N+1**

Mon 4/7	Tue 4/8	Wed 4/9	Thu 4/10	Fri 4/11
No lecture				**Recitation** Register Allocation
			Due: Mini-quiz, weekly check-in	

Lecture forecast... **Week N+2**

Mon 4/15	Tue 4/16	Wed 4/17	Thu 4/18	Fri 4/19
Lecture Dataflow Theory	**Lecture** Dataflow Theory	**No lecture**	**No lecture**	**Recitation** Phase 5 infosession
			Due: Mini-quiz, weekly check-in	**Due: Project phase 4**

Weekly updates

Phase 4 info ←

Project overview

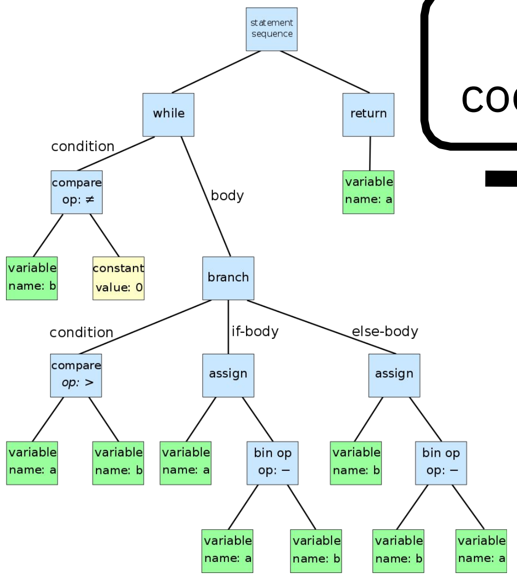
```
import printf;

void main() {
…
```

Decaf source file

Phase 1. Does it have the right structure? (syntax)

Phase 2. Does it make sense? (semantics)



Internal representation

Phase 3
code generation

```
push %rbp
mov %rsp, %rbp
…
```

x86-64 assembly

So we have a working compiler now…*

what next?

* Or by the end of today

Project overview

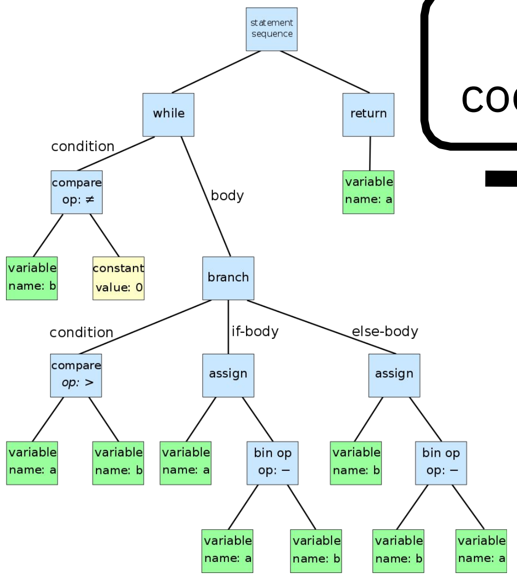
```
import printf;

void main() {
...
```

Decaf source file

Phase 1. Does it have the right structure? (syntax)

Phase 2. Does it make sense? (semantics)



Internal representation

Phase 3
code generation

```
push %rbp
mov %rsp, %rbp
...
```

Optimized
x86-64 assembly

Phase 4. What can we learn about the program? (dataflow analysis)

Project overview

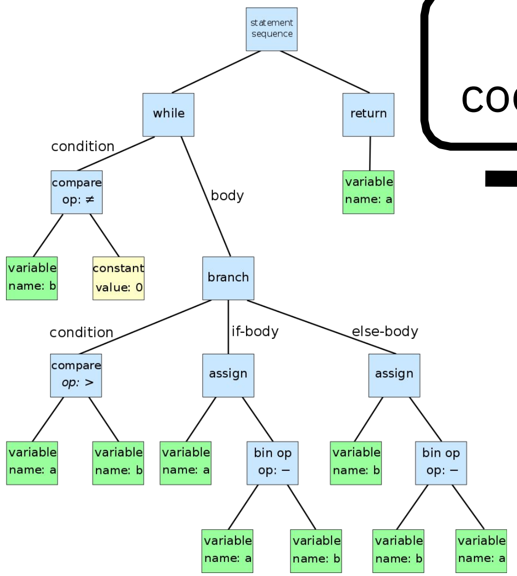
```
import printf;

void main() {
...
```

Decaf source file

Phase 1. Does it have the right structure? (syntax)

Phase 2. Does it make sense? (semantics)



Internal representation

Phase 3
code generation

Phase 4. What can we learn about the program? (dataflow analysis)

```
push %rbp
mov %rsp, %rbp
...
```

Even more optimized x86-64 assembly

Phase 5. How can we make the output code faster?

From now on, the project becomes more open-ended.

We'll require some specific optimizations, but other than that you are free to implement whatever your heart desire.

At the end of phase 5, there will be a **compiler derby** to find which team's compiler produces the fastest code!

Logistics and requirements

Phase 4 overview

- **Required:** implement **at least one of the following global dataflow optimizations**
  - Copy propagation
  - Common subexpression elimination
  - Dead code elimination
- Optimization should **at least work on statements involving local (non-array) variables**

Dataflow analysis: overview

- A form of **program analysis**: compile-time reasoning about program behavior
- Store **some information** we've learned about the program at each program point (CFG node)
- At each node, need to update information based on content of the node **("transfer function"),** and propagate information to successor nodes *(or predecessors for backwards analyses)*
- At merge points, need to **combine** information somehow
- Iterate until we reach a fixed point
- *More of this formalization in N+2 week's lectures!*

Copy propagation

- Propagate copies (assignments like a ← b)
- Based on **reaching definitions analysis:** which definitions of each variable reaches each program point*

`a ← b` `c ← a + 1`	`a ← b` `c ← b + 1`
Before	After

Copy propagation

- Be careful about this!

a ← b	a ← b
b ← c	b ← c
d ← a	d ← b ???

- One way to avoid: just keep track of which variables are copies of each other instead of using reaching definitions

Dead code elimination

- Remove code that computes variables that are not used
- Based on **liveness analysis:** which variables are “live” (has a use afterwards)

a ← x + y x ← a + b (a is global, x is local to method)	a ← x + y
Before	After

Common subexpression elimination

- Only compute an expression once
- Based on **Available expressions analysis:** which expressions defined earlier are still valid (operands not modified)

a. ← x + y b. ← x + y c. x ← a d. ← x + y	t1 ← x + y a. ← t1 b. ← t1 x ← a c. ← x + y
Before	After

Summary

Optimization	**Analysis**
Copy propagation	Reaching definitions* *be careful
Common subexpression elimination	Available expressions
Dead code elimination	Liveness

Summary

	Reaching Definitions	Live Variables	Available Expressions
Domain	Sets of definitions	Sets of variables	Sets of expressions
Direction	Forwards	Backwards	Forwards
Transfer function	$gen_B \cup (x - kill_B)$	$use_B \cup (x - def_B)$	$e\_gen_B \cup (x - e\_kill_B)$
Boundary	$\text{OUT}[\text{ENTRY}] = \emptyset$	$\text{IN}[\text{EXIT}] = \emptyset$	$\text{OUT}[\text{ENTRY}] = \emptyset$
Meet ($\wedge$)	$\cup$	$\cup$	$\cap$
Equations	$\text{OUT}[B] = f_B(\text{IN}[B])$ $\text{IN}[B] = \bigwedge_{P,pred(B)} \text{OUT}[P]$	$\text{IN}[B] = f_B(\text{OUT}[B])$ $\text{OUT}[B] = \bigwedge_{S,succ(B)} \text{IN}[S]$	$\text{OUT}[B] = f_B(\text{IN}[B])$ $\text{IN}[B] = \bigwedge_{P,pred(B)} \text{OUT}[P]$
Initialize	$\text{OUT}[B] = \emptyset$	$\text{IN}[B] = \emptyset$	$\text{OUT}[B] = U$

Figure 9.21: Summary of three data-flow problems

Phase 4 overview (cont'd)

- **Optional:** extend optimizations to global variables and array variables
- **Optional:** other optimizations (more info in handout)
  - Constant propagation and folding
  - Loop-invariant code motion
  - Unreachable code elimination
  - Algebraic simplification *(not dataflow)*
  - …

Submission and grading

- Phase 4 is worth **10%** of the overall grade, due **Friday, April 18.**
- Two items to be submitted on Gradescope
  - Design document (8%)
    - Overall dataflow framework (3%)
    - Details of implemented dataflow optimizations (4%)
    - Extras, difficulties, and contributions (1%)
  - Code submission, autograded on correctness only (2%)
    - No private test cases
    - Output code should be correct with and without optimizations

Specifications

- Your compiler should be **correct** with or without optimizations
- When running
  ./run.sh <filename> –t assembly
  on a valid input file:
  - Outputs x86-64 assembly code to the output file (or stdout if –o is not specified)
- We'll assemble using
  gcc -O0 -no-pie output.s -o output.exe

CLI for optimizations

- **-O cse** turns on common subexpression elimination only
- **-O dce** turns on dead code elimination only
- **-O cp,cse** turns on copy propagation and common subexpression elimination only
- **-O all turns on all optimizations** (we'll run the autograder with this option)
- **-O all,-cse** turns on all optimizations except common subexpression elimination

Design document

- Explains technical details
- Includes the following sections:
  1. Design *(including general dataflow framework and specific details for each implemented optimization)*
  2. Extras
  3. Difficulties
  4. Contribution

1. Design

- Overview of your design, including design choices you made and design alternatives you considered.
- This section should help us understand your code
- In particular, please include:
  - Your general framework for dataflow optimizations (worth **3%**)
  - Details of each dataflow optimization you implemented (worth **4%**, more info on next slide)

1. Design — details

- For each dataflow optimization you implemented, please include:
  - the scope of the optimization (did you take into account global variables and/or array variables?)
  - the dataflow equations you used
  - a sample test case, with generated code before and after, included under `doc/phase4-code/` in your repository
  - a brief explanation of how your dataflow optimization worked

Other sections (worth **1%**)

2. Extras:
   - Any clarifications, assumptions, or additions you made
   - Any interesting debugging techniques, build scripts
   - Anything cool you'd like to share!
3. Difficulties:
   - List of known problems with your project, and as much as you know about the cause
   - Any issues from phase 3 that you fixed
4. Contributions: A brief description of how your group divided the work

Words of advice

- **Start simple!**
  - Start with very simple test cases so that you understand what’s happening
  - Start with local non-array variables only, and only add global variables / array variables after you can get the analysis to work on local variables

- **Keep things general**
  - Various dataflow analyses can all be written in terms of a transfer function and a meet function
  - Consider making a parametrized dataflow framework
  - *Next week's lecture will cover this formalization*

	Reaching Definitions	Live Variables	Available Expressions
Domain	Sets of definitions	Sets of variables	Sets of expressions
Direction	Forwards	Backwards	Forwards
Transfer function	$gen_B \cup (x - kill_B)$	$use_B \cup (x - def_B)$	$e\_gen_B \cup (x - e\_kill_B)$
Boundary	$\text{OUT}[\text{ENTRY}] = \emptyset$	$\text{IN}[\text{EXIT}] = \emptyset$	$\text{OUT}[\text{ENTRY}] = \emptyset$
Meet ($\wedge$)	$\cup$	$\cup$	$\cap$
Equations	$\text{OUT}[B] = f_B(\text{IN}[B])$ $\text{IN}[B] = \bigwedge_{P,pred(B)} \text{OUT}[P]$	$\text{IN}[B] = f_B(\text{OUT}[B])$ $\text{OUT}[B] = \bigwedge_{S,succ(B)} \text{IN}[S]$	$\text{OUT}[B] = f_B(\text{IN}[B])$ $\text{IN}[B] = \bigwedge_{P,pred(B)} \text{OUT}[P]$
Initialize	$\text{OUT}[B] = \emptyset$	$\text{IN}[B] = \emptyset$	$\text{OUT}[B] = U$

Figure 9.21: Summary of three data-flow problems

- **Consider using single-statement blocks**
  - More time/memory-consuming but who cares
  - No need to propagate information inside a basic block
  - One tricky thing: Need to be able to add/remove nodes/merge points/join points.

- **Use array of nodes, not pointer-and-objects (especially for Rust teams)**
  - Key: Need to be able to remove/add statements
  - Especially relevant if you don't use basic blocks
  - You will need adjacency list and reverse adj. list

GDB crash course

Code available at:

https://github.com/6110-sp25/recitation9