6.110 Computer Language Engineering

Recitation 5: Phase 3 infosession

March 8, 2024

Weekly updates ←

Phase 3 info

Phase 3 demo

Wrapping up phase 2…

- **Project phase 2 is due today 11:59PM!!!**
  - **This includes the report!**
  - 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 3 has been released, due **Friday, April 5** (Friday after Spring Break)
- Miniquiz 5 and Weekly Check-in 6 are due Thursday, March 14 🥧
  - Reminder: these are graded on completion – please submit!!

Quiz 1: **Friday, March 15**

- Quiz will be in class, worth **10%** of the overall grade
- Covers lecture content up to yesterday's lecture:
  - Regex, context-free grammars
  - Top-down parsing
  - High-level IR and semantics
  - Unoptimized codegen
- Past quizzes are now on course website
- Quiz review session on **Wednesday, March 13** during re-lecture time (4-6pm in 26-322).

Coming up soon... **Week 6**

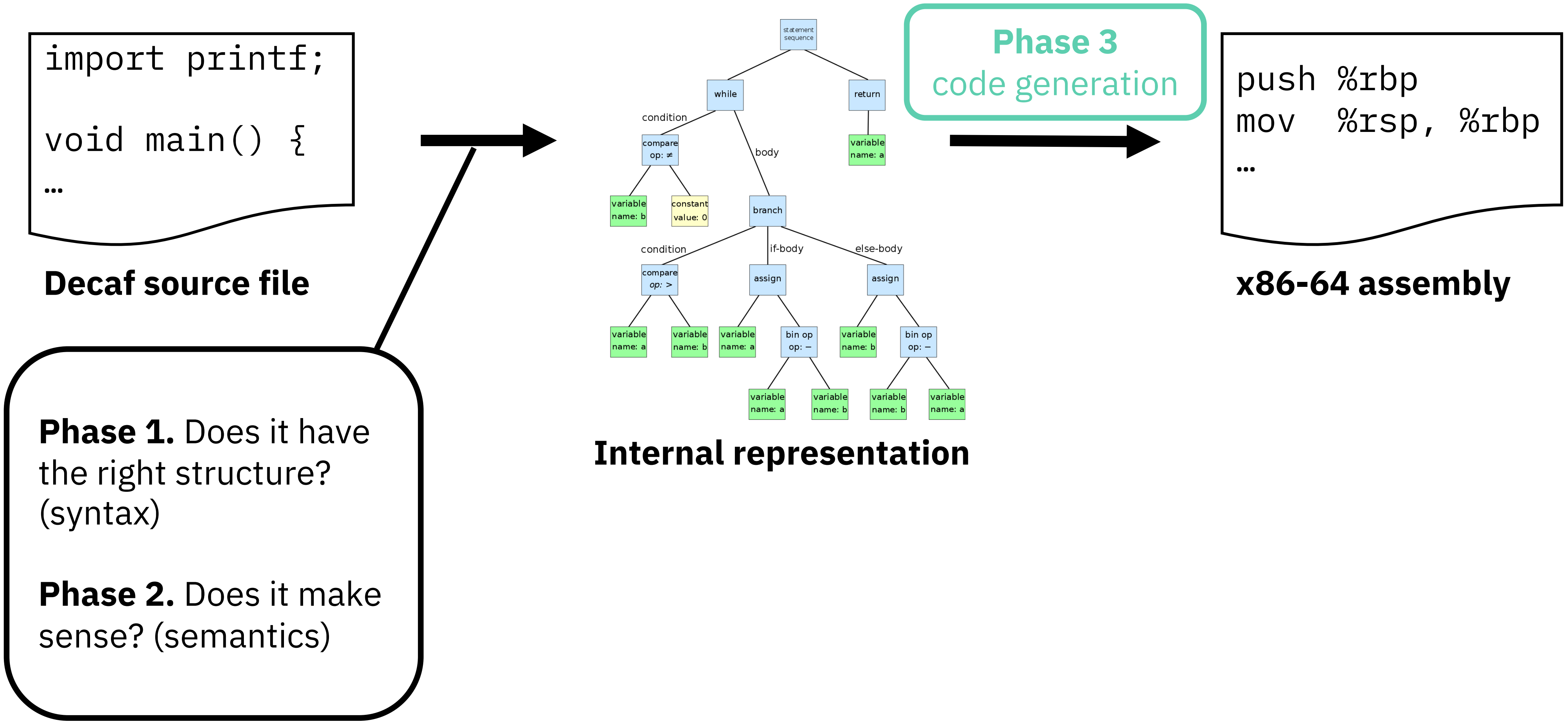
Mon 3/11	Tue 3/12	Wed 3/13	Thu 3/14	Fri 3/15
No lectures next week Lectures will tentatively resume Mon 3/18				**Quiz 1** up to Codegen lectures
		Quiz review (4-6pm in 26-322)	**Due:** Mini-quiz, weekly check-in	

Weekly updates

Phase 3 info ←

Phase 3 demo

Logistics and requirements



import printf;
void main() {
…
Decaf source file
Phase 1. Does it have the right structure? (syntax)
Phase 2. Does it make sense? (semantics)
statement sequence
while
return
condition
body
compare op: ≠
variable name: b
constant value: 0
variable name: a
branch
condition
if-body
else-body
compare op: >
assign
assign
variable name: a
variable name: b
variable name: a
bin op op: –
variable name: b
bin op op: –
variable name: a
variable name: b
variable name: b
variable name: a
Internal representation
Phase 3 code generation
push %rbp
mov %rsp, %rbp
…
x86-64 assembly

Phase 3 overview

- **Goal:** have a fully working compiler!
  - ***Unoptimized*** code generation – the goal is to be correct, not to be fast
- Work in same teams, same GitHub repo from Phase 2
  - If you’d like to name your team, it’s still not too late! Please let us know and we’ll change your repository name.

Submission and grading

- Phase 3 is worth **15%** of the overall grade, due **Friday, April 5.**
- Two items to be submitted on Gradescope
  - Code submission: **12%** (all autograded)
  - Design document: **3%**
    - Submit PDF on Gradescope *and* include in your repository.

Specifications

- 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`

Runtime checks

Your compiler should emit code that performs the following runtime checks:

1. Array bounds checking: array index must be in bounds (0 to length – 1, inclusive)
   (if out of bounds, should terminate with exit code **–1**)
2. Control must not fall off the end of a method that returns a value.
   (if control falls off, should terminate with exit code **–2**)

Design document

- Explains technical details of phases 1-3
- Around 5 pages long
- Includes the following sections:
    1. Design
    2. Extras
    3. Difficulties
    4. Contribution
- If you used LLMs, also describe how you used them and provide chat logs

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 these:
  - Explanation of the compilation steps (entry point, flow, etc)
  - Discussion of your designs for
    (i) high-level IR, (ii) semantic checker, (iii) low-level IR, and (iv) code generator

2. Extras

- Any clarifications, assumptions, or additions you made
- Any interesting debugging techniques, build scripts, or approved libraries
- Anything cool you’d like to share!

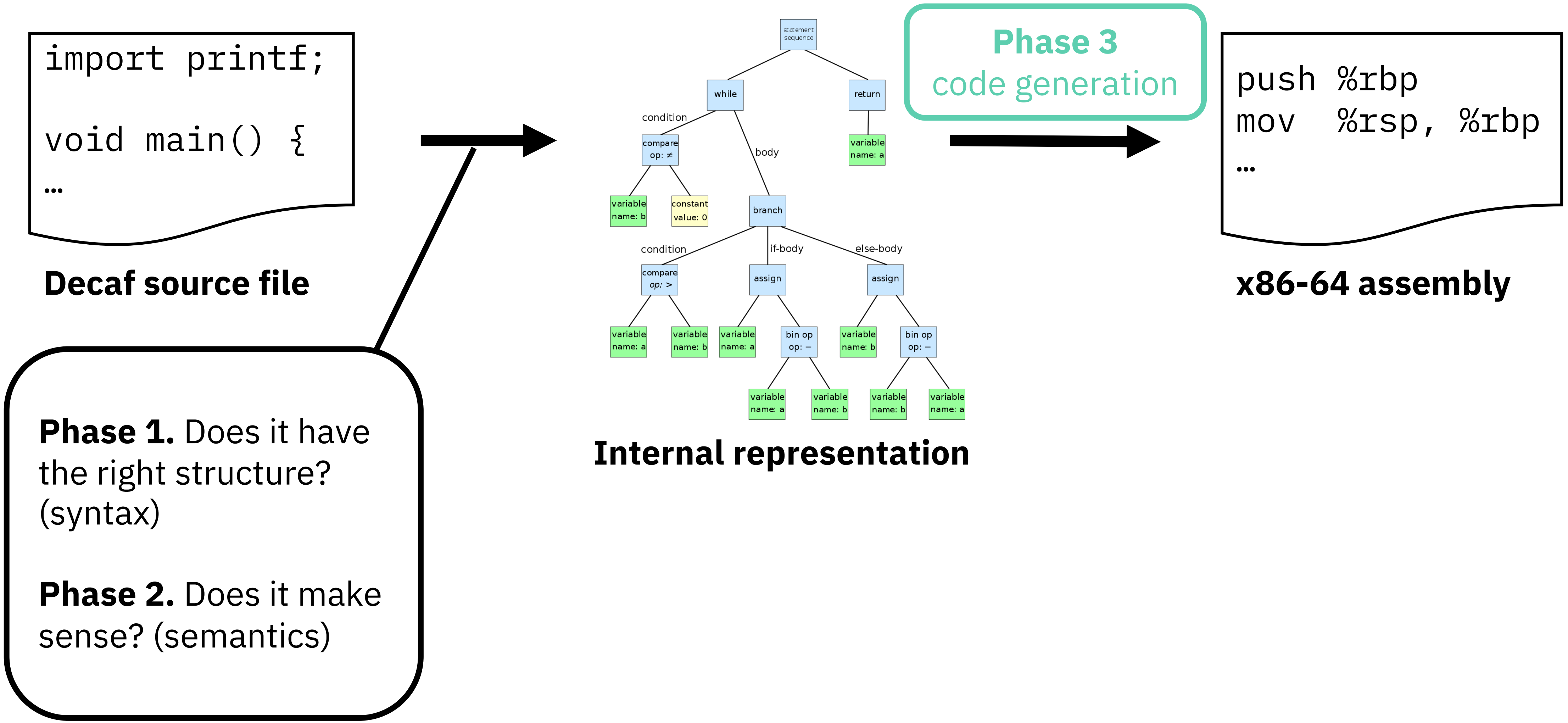
3. Difficulties

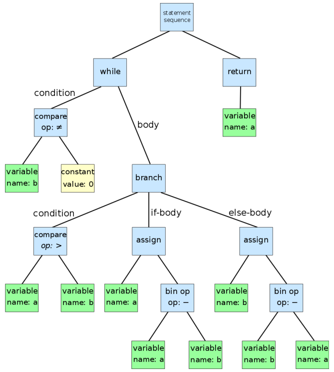
- List of known problems with your project, and as much as you know about the causes
- Any issues from phase 2 that you fixed

4. Contribution

- Brief description of how your team divided the work.
- (This will not affect your grade.)

Suggested approach


import printf;
void main() {
...
Decaf source file
Phase 1. Does it have the right structure? (syntax)
Phase 2. Does it make sense? (semantics)
statement sequence
while
return
condition
body
compare op: ≠
variable name: b
constant value: 0
branch
variable name: a
condition
if-body
else-body
compare op: >
assign
assign
variable name: a
variable name: b
variable name: a
bin op op: −
variable name: b
bin op op: −
variable name: a
variable name: b
variable name: b
variable name: a
Internal representation
Phase 3
code generation
push %rbp
mov %rsp, %rbp
...
x86-64 assembly



High-level IR (AST)

Structured control flow
if/else, loops, break, continue

Complex expressions
`x+=y[4*z]/a`

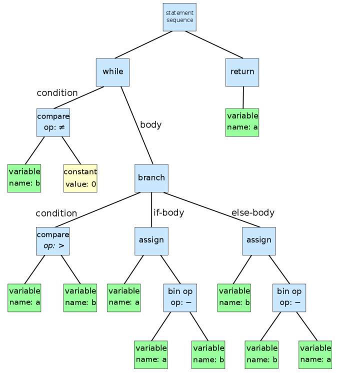
Phase 3
code generation

```
push %rbp
mov  %rsp, %rbp
…
```

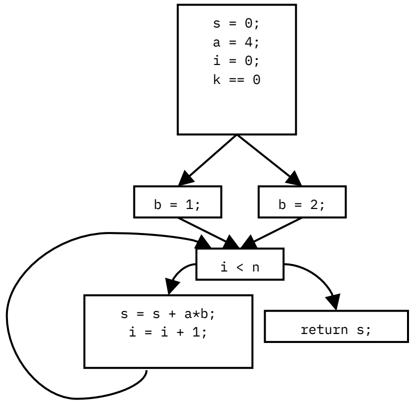
x86-64 assembly

Unstructured control flow
jumps only!

Two-address code
`mulq $4, %rcx`

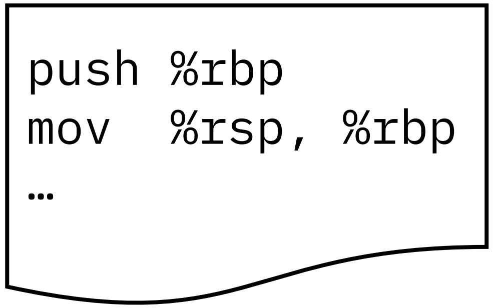


High-level IR (AST)



Low-level IR (CFG)

Code generation →



x86-64 assembly

Structured control flow
if/else, loops, break, continue

Destructuring →

Unstructured control flow
edges = jumps

Unstructured control flow
jumps only!

Complex expressions
`x+=y[4*z]/a`

Linearizing →

Three-address code
`t1 ← 4 * z`

Two-address code
`mulq $4, %rcx`

Note: The TAs recommend using a linearized CFG. This is different from what is shown in lecture slides, but will make code generation and optimizations in future phases a lot simpler!

CFG

- **Nodes** = computation
  - Can be **basic blocks**, i.e. list of instructions, where there are no branches in/out in the middle of a basic block
  - Another approach is to use **single-statement blocks**, i.e. treat every instruction as being in a separate block
- **Edges** = control flow

Destructuring control flow

- Keep in mind that Decaf has **short-circuiting boolean operations everywhere**
- As in codegen lecture slides, can be implemented using two recursive functions:
  - `destruct` : destructs statement-level control flow (if/else, for, while, break, continue)
  - `shortcircuit` : deals with expression-level control flow (&&, | |)

Linearizing expressions

- We suggest using a flat list approach
- Recursive function `linearize` (shortened as `L`)
  - Input: an expression node `(x op y)`
  - Output: a pair `(i,v)` where `i` is a list of three-address instructions, and `v` is the variable that stores the result
  - If `(ix,vx) = L(x)` and `(iy,vy) = L(y)` then `L(x op y) = (ix + iy + [z = vx op vy], z)`

Code generation

- General approach:
  - Allocate space for globals
  - Output code for each function separately. For each function, output prolog, then body, then epilog
- Use templates for each pattern (operation, control flow, etc.)
  - If you don’t know what to do, use Godbolt to see what gcc/clang does
- You’ll have to deal with various quirks of x86-64 assembly. Make sure to look at the x86-64 references.

printf

- **You need to support printf!**
- Decaf does not have I/O functions, so this is the only way we can test your code
- printf is special:
  - Stack (`%rsp`) needs to be 16-byte aligned when printf is called (or when any external function is called)
  - `%rax` needs to contain the number of floating point arguments (always 0 for Decaf code)
- Make sure you pass the public test cases with printf

exit codes

- **To set a nonzero exit code, use the `exit` syscall**
- In Linux (which is what the autograder is running), this is syscall number 60
- Use `syscall` in assembly
  - `%rax` should be set to 60 (syscall number)
  - `%rdi` (first argument) should be set to exit code

General words of advice

- **Start early!**
  - We give you a lot of time for this phase because it’s usually the longest phase.
- **Have regular team meetings.**
  - From our experience, it’s easiest to get things done if you’re working all together in person. Pick a regular time and place for meetings and stick to it!

General words of advice

- **Do the simplest thing.**
  - Don’t worry about performance of the generated code. You’ll have the entire second half of the semester to do optimizations.
- **Keep abstraction level consistent.**
  - It’s fine (maybe even a good idea) to have many IRs and many passes through IRs, but try to keep each IR self-consistent!

Weekly updates

Phase 3 info

Phase 3 demo ←

Phase 3 demo

Code available at:

https://github.com/6110-sp24/recitation5