

Code Generation

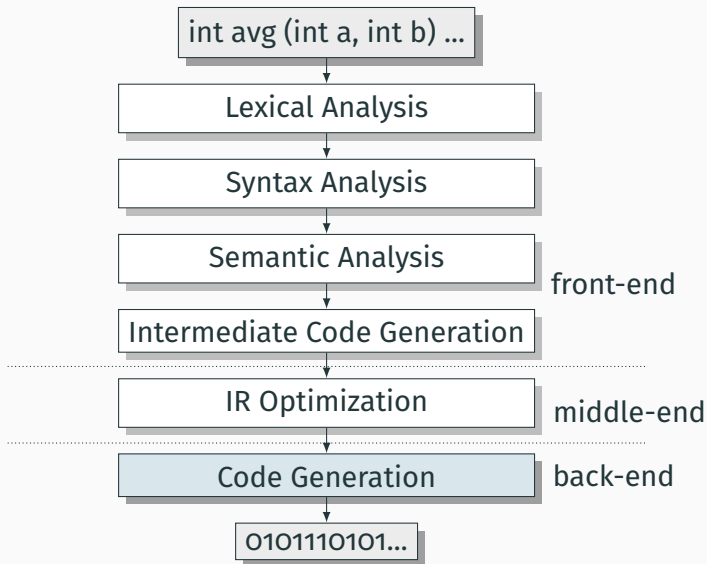
Ronghui Gu

Spring 2020

Columbia University

* Course website: <https://www.cs.columbia.edu/~rgu/courses/4115/spring2019>

Code Generation



- Choose the appropriate machine instructions for each IR instruction.
- Manage **finite** machine resources (e.g., registers).
- Implement runtime environment.

The Memory Hierarchy

Memory tradeoffs: there is an enormous tradeoff between **speed** and **size** in memory.

Register Allocation

Using registers intelligently is a critical step in any compiler.

Explore three algorithms for register allocation:

- Naive (“no”) register allocation.
- **Linear scan** register allocation.
- **Graph-coloring** register allocation.

Naive Register Allocation.

Idea: store every value in main memory, loading values only when they're needed.

- Insert **load** to pull the values from memory into registers before access.
- Insert **store** to store the values back into memory after access.

```
a = b + c;
```

```
d = a;
```

A Better Allocator

Goal: try to hold as many variables in registers as possible.

Register consistency:

- At each program point, each variable must be in the same location.
- At each program point, each register holds at most one **live** variable.

Live Intervals

Live interval: the smallest subrange of the IR code containing all a variable's live ranges.

`e = d + a;`

`f = b + c;`

`f = f + b;`

`d = e + f;`

`g = d;`

Live Intervals

Live interval: the smallest subrange of the IR code containing all a variable's live ranges.

{ d, b, c, a }

e = d + a;

{ e, b, c }

f = b + c;

{ e, f, b }

f = f + b;

{ e, f }

d = e + f;

{ d }

g = d;

{ g }

The register Interference Graph (RIG)

{ d, b, c, a }

e = d + a;

{ e, b, c }

f = b + c;

{ e, f, b }

f = f + b;

{ e, f }

d = e + f;

{ d }

g = d;

{ g }