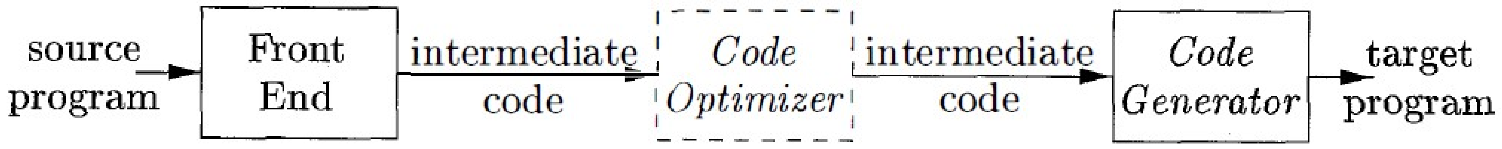
CSE504 Compiler Design
Code Generation

YoungMin Kwon

Code Generation

- Takes an intermediate representation of the source.
  - Postfix, Three-address code, Stack machine code, Syntax Tree, Dags
- Produces an equivalent target program.
  - Absolute machine code, Relocatable machine code, Assembly code



Issues to Consider

- Memory management
  - Type (width) and Offset info are in the Symbol Tables.
  - Conversion from labels to addresses is analogous to the backpatching technique
- Instruction Selection
  - Remove redundancy
  - Consider special instructions (inc for add 1)

```
a := b + c    mov b, r0             mov a, r0
d := a + e    add c, r0             add 1, r0
              mov r0, a             mov r0, a
              mov a, r0
              add e, r0             inc a
              mov r0, d
```

Issues to Consider

- Register Allocation
  - Register allocation: select the set of variables that will reside in registers.
  - Register assignment: pick the specific register for a variable.
  - Register usage conventions, Register pairs
    - E.g. mul (rax, rdx), div (rax, rdx), loop (rcx register)
- Evaluation Order
  - Some orders require fewer registers than others

Runtime Storage Management

- Static Allocation (e.g. call, return)

```
mov #here+20, callee.static_area
goto callee.code_area
...
goto *callee.satic_area
```

- Stack Allocation (e.g. call, return)

```
add #caller.recordsize, sp
mov #here+16, *sp
goto callee.code_area
...
goto *sp
sub #caller.recordsize, sp
```

Flow Graphs

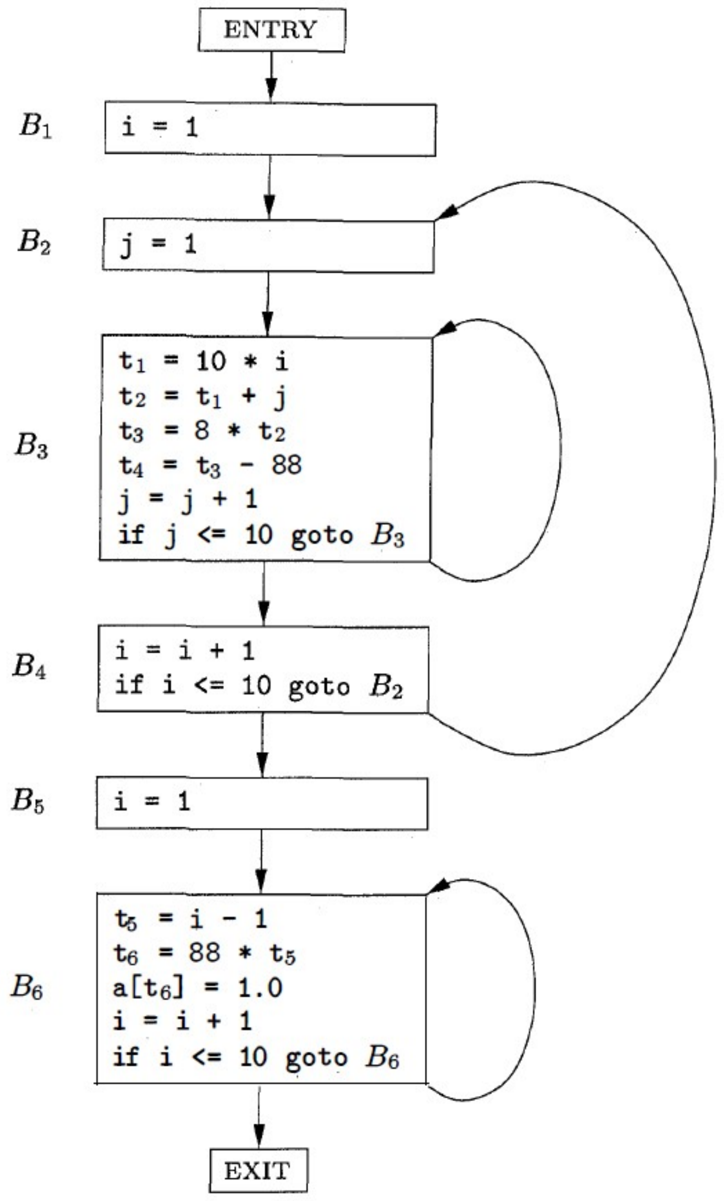
- Basic Blocks
  - Determine the set of **leaders**
    - The first statement is a leader
    - Any target of a goto (conditional or unconditional) is a leader
    - Any statement immediately following goto (conditional or unconditional) is a leader
  - For each leader, its **basic block** consists of the leader and all statements up to but not including the next leader.

Flow Graphs

- Flow graphs are a directed graph
  - Nodes: basic blocks
  - Edges: there is an edge from B1 to B2 if
    - There is a jump from the last statement of B1 to the first statement of B2
    - B2 immediately follows B1 and B1 does not end with an unconditional jump

```
**for** $i$ from 1 to 10 **do**
    **for** $j$ from 1 to 10 **do**
        $a[i, j] = 0.0;$
**for** $i$ from 1 to 10 **do**
    $a[i, i] = 1.0;$
```

```
 1)  i = 1
 2)  j = 1
 3)  t1 = 10 * i
 4)  t2 = t1 + j
 5)  t3 = 8 * t2
 6)  t4 = t3 - 88
 7)  a[t4] = 0.0
 8)  j = j + 1
 9)  if j <= 10 goto (3)
10)  i = i + 1
11)  if i <= 10 goto (2)
12)  i = 1
13)  t5 = i - 1
14)  t6 = 88 * t5
15)  a[t6] = 1.0
16)  i = i + 1
17)  if i <= 10 goto (13)
```



```
B1:  i = 1

B2:  j = 1

B3:  t1 = 10 * i
     t2 = t1 + j
     t3 = 8 * t2
     t4 = t3 - 88
     j = j + 1
     if j <= 10 goto B3

B4:  i = i + 1
     if i <= 10 goto B2

B5:  i = 1

B6:  t5 = i - 1
     t6 = 88 * t5
     a[t6] = 1.0
     i = i + 1
     if i <= 10 goto B6
```

Flow Graphs

- Loop: a collection of nodes such that
  - All nodes in the collection is strongly connected
  - Has a unique entry (nodes outside of the loop has to go through the entry to reach any inside nodes)
- Inner loop
  - A loop that contains no other loop.

Next-Use Information

- Next-Use: Where will the computed variable be used
- Computing Next-Use:
  For each i: x := y op z found during the backward scan
  - Attach the next-use and liveness info about x, y, z to statement i.
  - In the symbol table set x to “not live”, “no next use”
  - In the symbol table set y and z to “live”, and i for the next use.
- Pack two temporaries into the same location if they are not live simultaneously.

Simple Code Generator

- Register descriptor for each register
  - Keeps track of the variable names whose value is in that register.
- Address descriptor for each variable
  - Keeps track of the locations (register, memory, stack, …) where the variable can be found.

Code-Generation Algorithm

- For three-address instructions x = y + z
  - Use getReg(x=y+z) to select registers for x, y, z. Call them Rx, Ry, and Rz.
  - If y is not in Ry (by the register descriptor for Ry), then issue LD Ry, y’, where y’ is one of the memory location for y (by the address descriptor for y)
  - Action for z is analogous to y
  - Issue the instruction Add Rx, Ry, Rz
- For copy statements x = y
  - Assume getReg(x = y) will choose the same register for x and y.
  - If y is not in Ry, issue Ld Ry, y
  - If y was already in Ry, do nothing
  - Adjust the descriptors such that Ry has x

Register and Address Descriptors

- For the instruction Ld R, x
  - Make the register descriptor for R hold only x
  - Add R to the address descriptor for x
- For the instruction ST x, R
  - Add the memory location for x to the address descriptor for x
- For the instructions ADD Rx, Ry, Rz
  - Make the register descriptor for Rx hold only x
  - Make the address descriptor for x hold only Rx
  - Remove Rx from address descriptors of any variables other than x
- For the copy statement x = y
  - Add x to the register descriptor for Ry
  - Make the address descriptor for x hold only Ry

```
t = a - b
u = a - c
v = t + u
a = d
d = v + u
```

R1	R2	R3	a	b	c	d	t	u	v
			a	b	c	d			

```
t = a - b
    LD R1, a
    LD R2, b
    SUB R2, R1, R2
```

R1	R2	R3	a	b	c	d	t	u	v
a	t		a, R1	b	c	d	R2		

```
u = a - c
    LD R3, c
    SUB R1, R1, R3
```

R1	R2	R3	a	b	c	d	t	u	v
u	t	c	a	b	c, R3	d	R2	R1	

```
v = t + u
    ADD R3, R2, R1
```

R1	R2	R3	a	b	c	d	t	u	v
u	t	v	a	b	c	d	R2	R1	R3

```
a = d
    LD R2, d
```

R1	R2	R3	a	b	c	d	t	u	v
u	a, d	v	R2	b	c	d, R2		R1	R3

```
d = v + u
    ADD R1, R3, R1
```

R1	R2	R3	a	b	c	d	t	u	v
d	a	v	R2	b	c	R1			R3

```
exit
    ST a, R2
    ST d, R1
```

R1	R2	R3	a	b	c	d	t	u	v
d	a	v	a, R2	b	c	d, R1			R3

getReg(x=y+z)

- If y is in a register, pick the register as Ry
- Else if there is an empty register, pick the register as Ry
- Else
  - Let R be a candidate register and v be a variable in its register descriptor: (we need to store the value of R to the memory for each v)
  - If the address descriptor of v has other location than R, we are OK
  - If v is x and x is not an operand, we are OK
  - If v is not used later, we are OK
  - If we are not OK, we need to issue ST v, R
  - For each v in R’s register descriptor, find how many store operation is necessary and choose R that requires the least number of store operations.

getReg(x=y+z)

- If x is in a register that holds only x, pick the register as Rx
- Else if y is not used later and Ry holds only y, use Ry as Rx
- Else if there is an empty register, pick the register as Rx
- Else
  - Let R be a candidate register and v be a variable in its register descriptor: (we need to store the value of R to the memory for each v)
  - If the address descriptor of v has other location than R, we are OK
  - If v is x and x is not an operand, we are OK
  - If v is not used later, we are OK
  - If we are not OK, we will need to issue ST v, R
  - For each v in R's register descriptor, find how many store operation is necessary and choose R that requires the least number of store operations.
- For getReg(x=y), always choose Ry as Rx

Peephole Optimization

- Eliminating redundant Loads and Stores

```
LD R0, a
ST a, R0
```

- Eliminating Unreachable Code
  - Unreachable code: unlabeled code immediately following unconditional jump

```
If debug != 1 goto L2
;print debug information
L2:
```

Peephole Optimization

- Flow of Control optimizations
  - Jump to jump

```
    goto L1
    ...
L1: goto L2
```

```
    goto L2
    ...
L1: goto L2
```

  - Jump to conditional jump

```
    if a < b goto L1
    ...
L1: goto L2
```

```
    if a < b goto L2
    ...
L1: goto L2
```

  - Conditional jump to jump

```
    goto L1
    ...
L1: if a < b goto L2
L3:
```

```
L1: if a < b goto L2
    goto L3
    ...
L3:
```