Code Generation

Uday Khedker

(www.cse.iitb.ac.in/˜uday)

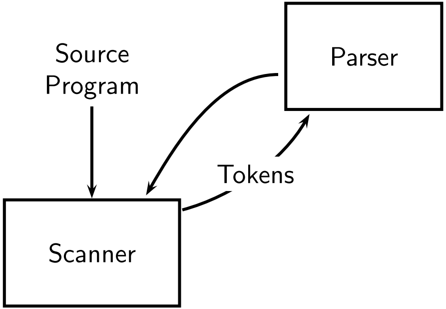
Department of Computer Science and Engineering,
Indian Institute of Technology, Bombay

March 2025

Typical Front Ends

Parser

Typical Front Ends



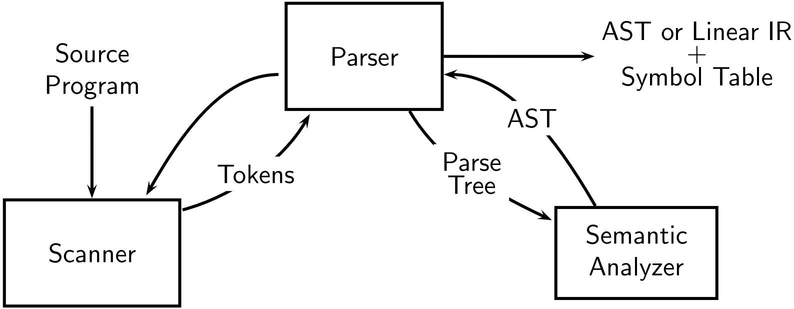
Source Program

Parser

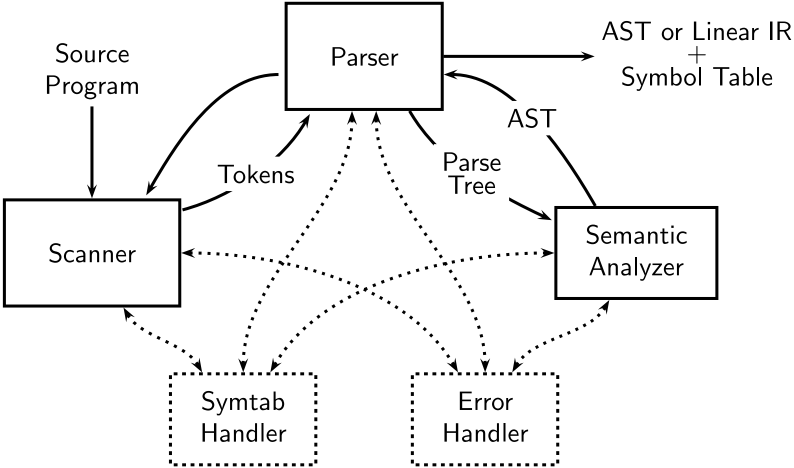
Tokens

Scanner

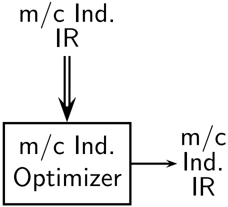
Typical Front Ends



Typical Front Ends



Typical Back Ends in Aho-Ullman Model



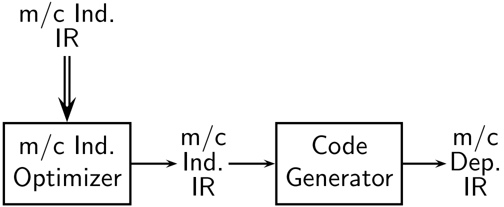
m/c Ind. IR

m/c Ind. Optimizer

m/c Ind. IR

- Compile time evaluations
- Eliminating redundant computations

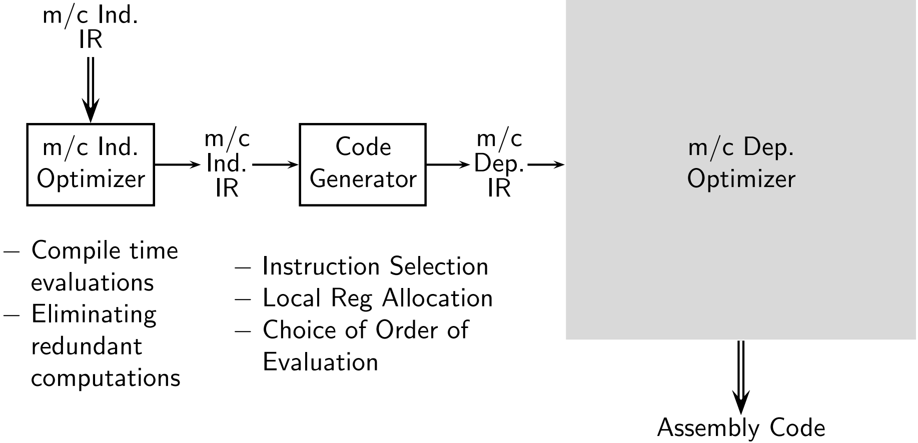
Typical Back Ends in Aho-Ullman Model



- Compile time evaluations
- Eliminating redundant computations

- Instruction Selection
- Local Reg Allocation
- Choice of Order of Evaluation

Typical Back Ends in Aho-Ullman Model



m/c Ind. IR

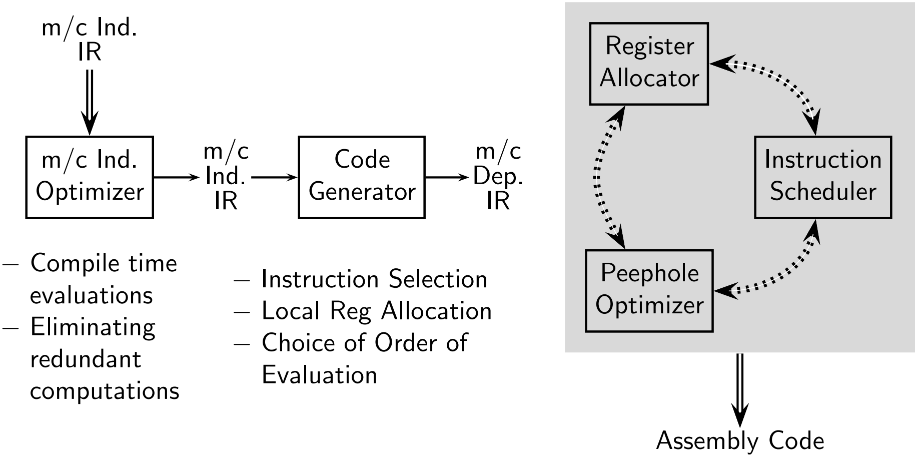
m/c Ind. Optimizer → m/c Ind. IR → Code Generator → m/c Dep. IR → m/c Dep. Optimizer

Assembly Code

- Compile time evaluations
- Eliminating redundant computations

- Instruction Selection
- Local Reg Allocation
- Choice of Order of Evaluation

Typical Back Ends in Aho-Ullman Model



- Compile time evaluations
- Eliminating redundant computations

- Instruction Selection
- Local Reg Allocation
- Choice of Order of Evaluation

Code Generation

- Register Allocation — We will cover this
- Instruction Selection — We will cover this
- Instruction Scheduling
- Peephole optimization

Global Register Allocation

Register Allocation

- Accessing values from registers is much faster than accessing them from memory

  Latencies for Intel Haswell in terms of cycles

  Register 1, L1 cache 4/5, L2 cache 12, L3 cache 36, RAM 264

Register Allocation

- Accessing values from registers is much faster than accessing them from memory

  Latencies for Intel Haswell in terms of cycles

  Register 1, L1 cache 4/5, L2 cache 12, L3 cache 36, RAM 264

- Issues
  - The number of registers is very small and the number of variables is large
  - Which variables should have their values in registers?
  - In which region of the program should the values be kept in registers?

Register Allocation

- Accessing values from registers is much faster than accessing them from memory

  Latencies for Intel Haswell in terms of cycles

  Register 1, L1 cache 4/5, L2 cache 12, L3 cache 36, RAM 264

- Issues
  - The number of registers is very small and the number of variables is large
  - Which variables should have their values in registers?
  - In which region of the program should the values be kept in registers?
- Categories
  - Local register allocation
    - Using registers to hold intermediate values of expressions
    - Usually, instruction selection algorithms handle this

      We will cover this
  - Global register allocation
    - Keeping the values in registers across statements

      We will cover this

Global Register Allocation Using Graph Colouring

Most popular approach

- - Identify live ranges
  - Construct interference graph
  - Colour the graph

Global Register Allocation Using Graph Colouring

Most popular approach

-
    - Identify live ranges
    - Construct interference graph
    - Colour the graph
- NP-complete in general
    - Excellent heuristics exists
    - We will study Chaitin-Briggs allocator

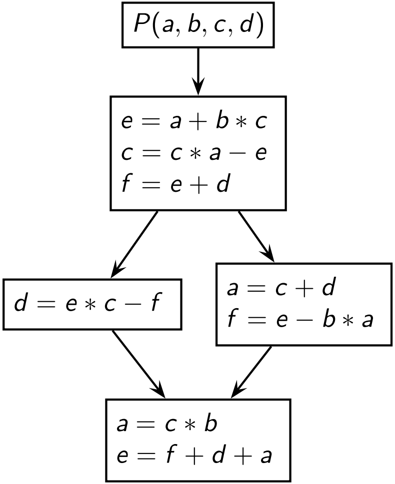
Global Register Allocation Using Graph Colouring

Most popular approach

-
    - Identify live ranges
    - Construct interference graph
    - Colour the graph
- NP-complete in general
    - Excellent heuristics exists
    - We will study Chaitin-Briggs allocator
- Decidable for chordal graphs

  Every cycle of length 4 or more has a chord connecting two nodes with an edge that is not part of the cycle (applies recursively)
    - Most practical interference graphs are chordal
    - All interference graphs for SSA representation are chordal

Motivating Example for Register Allocation



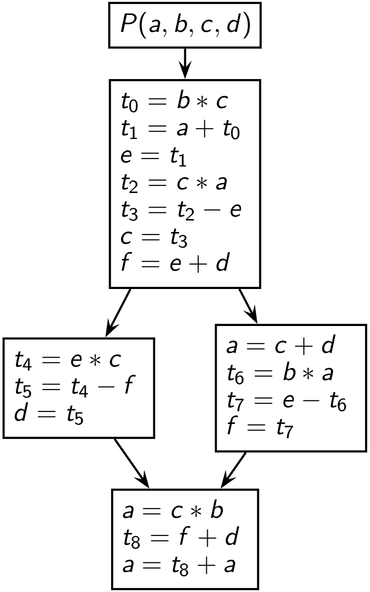
$P(a, b, c, d)$

$e = a + b * c$
$c = c * a - e$
$f = e + d$

$d = e * c - f$

$a = c + d$
$f = e - b * a$

$a = c * b$
$e = f + d + a$



$P(a, b, c, d)$

$t_0 = b * c$
$t_1 = a + t_0$
$e = t_1$
$t_2 = c * a$
$t_3 = t_2 - e$
$c = t_3$
$f = e + d$

$t_4 = e * c$
$t_5 = t_4 - f$
$d = t_5$

$a = c + d$
$t_6 = b * a$
$t_7 = e - t_6$
$f = t_7$

$a = c * b$
$t_8 = f + d$
$a = t_8 + a$

Steps in Chaitin-Briggs Register Allocator

1. Coalescing
    - Eliminating copy statements $x = y$ so that same register can be used for both x and y
    - We use copy propagation optimization for the purpose that replaces uses of x by y and eliminate the copy statement $x = y$
2. Identification of live ranges
    - Sequences of statements from a definition of a variable to its last use of that value
    - We use live variables analysis for the purpose
3. Identification of interference and construction of interference graph

    Live ranges l_1 and l_2 interfere if a definition of the variable of l_1 occurs in l_2 or vice-versa
4. Simplification of interference graph to identify the order in which the nodes should be coloured
5. Colouring the nodes

Copy Propagation Optimization (1)

- Perform reaching definition analysis
  - Assignment $\boxed{n: \ x = RHS}$ gives rise to definition x_n
  - Compute the sets of definition reaching every statement in the procedure
  - If definition x_n reaches assignment $\boxed{m: \ x = \ldots}$, then x_n is *killed* and a new definition x_m is *generated*

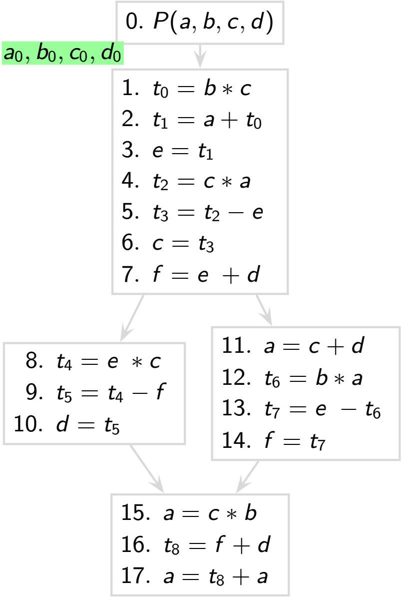
  When a definition x_n reaches some statement m, it suggests the existence of a control flow path from statement n to statement m along which x is not modified
- Set up the data flow equations over the control flow graph and compute the least fixed point solution
- This amounts to compute the def-use (and use-def) chains in a program

Copy Propagation Optimization (2)

- A use of x in statement m undergoes copy propagation if
  - a single definition $\boxed{n:\ x = RHS}$ reaches statement m, and
  - RHS is a variable and not an expression, and
  - the RHS variable is not modified along any path from statement n to statement m

  The use of x is replaced by the RHS variable
- After copy propagation, the assignment $\boxed{n:\ x = RHS}$ becomes dead and can be removed

Copy Propagation Optimization in Our Program



0. $P(a, b, c, d)$

a_0, b_0, c_0, d_0

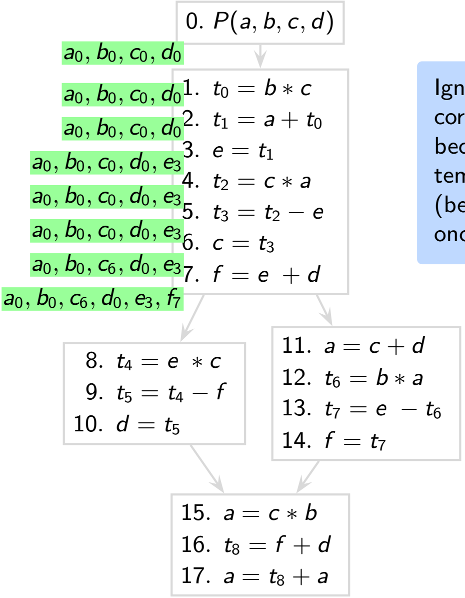
1. $t_0 = b * c$
2. $t_1 = a + t_0$
3. $e = t_1$
4. $t_2 = c * a$
5. $t_3 = t_2 - e$
6. $c = t_3$
7. $f = e + d$

8. $t_4 = e * c$
9. $t_5 = t_4 - f$
10. $d = t_5$

11. $a = c + d$
12. $t_6 = b * a$
13. $t_7 = e - t_6$
14. $f = t_7$

15. $a = c * b$
16. $t_8 = f + d$
17. $a = t_8 + a$

Copy Propagation Optimization in Our Program



0. $P(a, b, c, d)$

a_0, b_0, c_0, d_0

a_0, b_0, c_0, d_0 1. $t_0 = b * c$
a_0, b_0, c_0, d_0 2. $t_1 = a + t_0$
a_0, b_0, c_0, d_0, e_3 3. $e = t_1$
a_0, b_0, c_0, d_0, e_3 4. $t_2 = c * a$
a_0, b_0, c_0, d_0, e_3 5. $t_3 = t_2 - e$
a_0, b_0, c_6, d_0, e_3 6. $c = t_3$
7. $f = e + d$
$a_0, b_0, c_6, d_0, e_3, f_7$

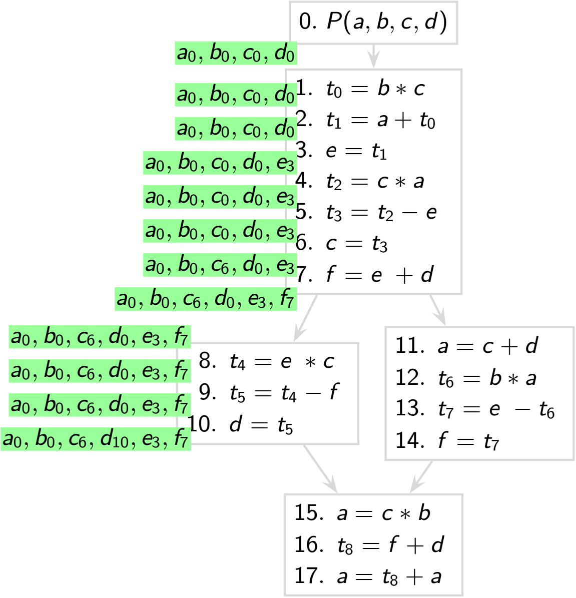
8. $t_4 = e * c$
9. $t_5 = t_4 - f$
10. $d = t_5$

11. $a = c + d$
12. $t_6 = b * a$
13. $t_7 = e - t_6$
14. $f = t_7$

15. $a = c * b$
16. $t_8 = f + d$
17. $a = t_8 + a$

Ignoring the definitions corresponding to the temporaries because connecting the definitions of temporaries to their usage is trivial (because a temporary is defined only once).

Copy Propagation Optimization in Our Program



0. $P(a, b, c, d)$

a_0, b_0, c_0, d_0

a_0, b_0, c_0, d_0 1. $t_0 = b * c$

a_0, b_0, c_0, d_0 2. $t_1 = a + t_0$

a_0, b_0, c_0, d_0, e_3 3. $e = t_1$

a_0, b_0, c_0, d_0, e_3 4. $t_2 = c * a$

a_0, b_0, c_0, d_0, e_3 5. $t_3 = t_2 - e$

a_0, b_0, c_6, d_0, e_3 6. $c = t_3$

7. $f = e + d$

$a_0, b_0, c_6, d_0, e_3, f_7$

$a_0, b_0, c_6, d_0, e_3, f_7$

$a_0, b_0, c_6, d_0, e_3, f_7$ 8. $t_4 = e * c$

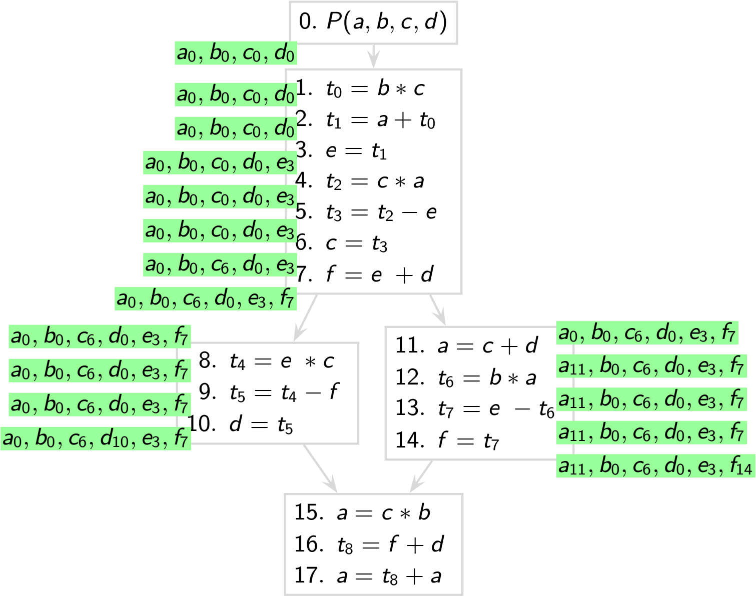
$a_0, b_0, c_6, d_0, e_3, f_7$ 9. $t_5 = t_4 - f$

$a_0, b_0, c_6, d_{10}, e_3, f_7$ 10. $d = t_5$

11. $a = c + d$
12. $t_6 = b * a$
13. $t_7 = e - t_6$
14. $f = t_7$

15. $a = c * b$
16. $t_8 = f + d$
17. $a = t_8 + a$

Copy Propagation Optimization in Our Program



0. $P(a, b, c, d)$

a_0, b_0, c_0, d_0

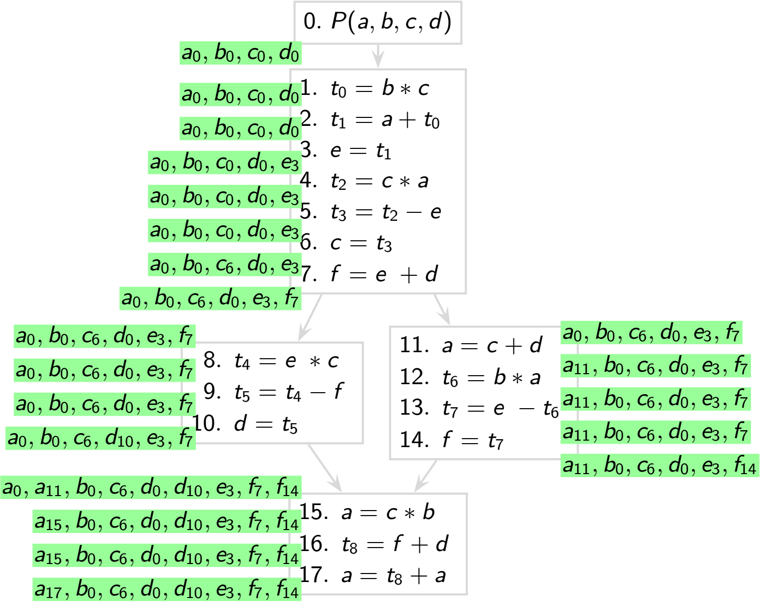
a_0, b_0, c_0, d_0 1. $t_0 = b * c$
a_0, b_0, c_0, d_0 2. $t_1 = a + t_0$
a_0, b_0, c_0, d_0, e_3 3. $e = t_1$
a_0, b_0, c_0, d_0, e_3 4. $t_2 = c * a$
a_0, b_0, c_0, d_0, e_3 5. $t_3 = t_2 - e$
a_0, b_0, c_6, d_0, e_3 6. $c = t_3$
$a_0, b_0, c_6, d_0, e_3, f_7$ 7. $f = e + d$

$a_0, b_0, c_6, d_0, e_3, f_7$ 8. $t_4 = e * c$
$a_0, b_0, c_6, d_0, e_3, f_7$ 9. $t_5 = t_4 - f$
$a_0, b_0, c_6, d_0, e_3, f_7$ 10. $d = t_5$
$a_0, b_0, c_6, d_{10}, e_3, f_7$

11. $a = c + d$ $a_0, b_0, c_6, d_0, e_3, f_7$
12. $t_6 = b * a$ $a_{11}, b_0, c_6, d_0, e_3, f_7$
13. $t_7 = e - t_6$ $a_{11}, b_0, c_6, d_0, e_3, f_7$
14. $f = t_7$ $a_{11}, b_0, c_6, d_0, e_3, f_7$
$a_{11}, b_0, c_6, d_0, e_3, f_{14}$

15. $a = c * b$
16. $t_8 = f + d$
17. $a = t_8 + a$

Copy Propagation Optimization in Our Program



0. $P(a, b, c, d)$

a_0, b_0, c_0, d_0

a_0, b_0, c_0, d_0
1. $t_0 = b * c$
a_0, b_0, c_0, d_0
2. $t_1 = a + t_0$
3. $e = t_1$
a_0, b_0, c_0, d_0, e_3
4. $t_2 = c * a$
a_0, b_0, c_0, d_0, e_3
5. $t_3 = t_2 - e$
a_0, b_0, c_0, d_0, e_3
6. $c = t_3$
a_0, b_0, c_6, d_0, e_3
7. $f = e + d$
$a_0, b_0, c_6, d_0, e_3, f_7$

$a_0, b_0, c_6, d_0, e_3, f_7$
8. $t_4 = e * c$
$a_0, b_0, c_6, d_0, e_3, f_7$
9. $t_5 = t_4 - f$
$a_0, b_0, c_6, d_0, e_3, f_7$
10. $d = t_5$
$a_0, b_0, c_6, d_{10}, e_3, f_7$

$a_0, b_0, c_6, d_0, e_3, f_7$
11. $a = c + d$
$a_{11}, b_0, c_6, d_0, e_3, f_7$
12. $t_6 = b * a$
$a_{11}, b_0, c_6, d_0, e_3, f_7$
13. $t_7 = e - t_6$
$a_{11}, b_0, c_6, d_0, e_3, f_7$
14. $f = t_7$
$a_{11}, b_0, c_6, d_0, e_3, f_{14}$

$a_0, a_{11}, b_0, c_6, d_0, d_{10}, e_3, f_7, f_{14}$
15. $a = c * b$
$a_{15}, b_0, c_6, d_0, d_{10}, e_3, f_7, f_{14}$
16. $t_8 = f + d$
$a_{15}, b_0, c_6, d_0, d_{10}, e_3, f_7, f_{14}$
17. $a = t_8 + a$
$a_{17}, b_0, c_6, d_0, d_{10}, e_3, f_7, f_{14}$

Copy Propagation Optimization in Our Program

0. $P(a, b, c, d)$

a_0, b_0, c_0, d_0

a_0, b_0, c_0, d_0 1. $t_0 = b * c$
a_0, b_0, c_0, d_0 2. $t_1 = a + t_0$
a_0, b_0, c_0, d_0, e_3 3. $e = t_1$
a_0, b_0, c_0, d_0, e_3 4. $t_2 = c * a$
a_0, b_0, c_0, d_0, e_3 5. $t_3 = t_2 - e$
a_0, b_0, c_6, d_0, e_3 6. $c = t_3$
7. $f = e + d$
$a_0, b_0, c_6, d_0, e_3, f_7$

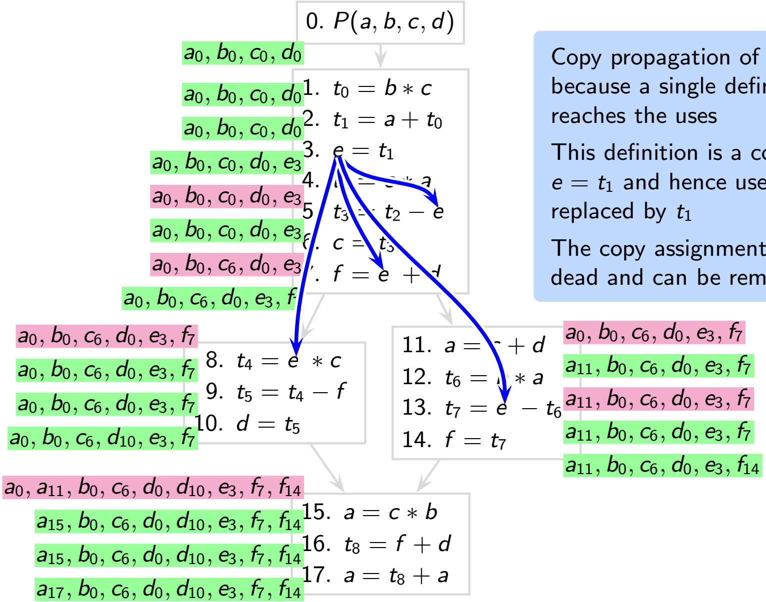
$a_0, b_0, c_6, d_0, e_3, f_7$
$a_0, b_0, c_6, d_0, e_3, f_7$ 8. $t_4 = e * c$
$a_0, b_0, c_6, d_0, e_3, f_7$ 9. $t_5 = t_4 - f$
10. $d = t_5$
$a_0, b_0, c_6, d_{10}, e_3, f_7$

$a_0, b_0, c_6, d_0, e_3, f_7$
11. $a = c + d$ $a_{11}, b_0, c_6, d_0, e_3, f_7$
12. $t_6 = b * a$ $a_{11}, b_0, c_6, d_0, e_3, f_7$
13. $t_7 = e - t_6$ $a_{11}, b_0, c_6, d_0, e_3, f_7$
14. $f = t_7$
$a_{11}, b_0, c_6, d_0, e_3, f_{14}$

$a_0, a_{11}, b_0, c_6, d_0, d_{10}, e_3, f_7, f_{14}$
15. $a = c * b$
$a_{15}, b_0, c_6, d_0, d_{10}, e_3, f_7, f_{14}$ 16. $t_8 = f + d$
$a_{15}, b_0, c_6, d_0, d_{10}, e_3, f_7, f_{14}$ 17. $a = t_8 + a$
$a_{17}, b_0, c_6, d_0, d_{10}, e_3, f_7, f_{14}$

We are interested in the highlighted data flow values.

Copy Propagation Optimization in Our Program



0. $P(a, b, c, d)$

a_0, b_0, c_0, d_0

a_0, b_0, c_0, d_0 1. $t_0 = b * c$

a_0, b_0, c_0, d_0 2. $t_1 = a + t_0$

a_0, b_0, c_0, d_0, e_3 3. $e = t_1$

a_0, b_0, c_0, d_0, e_3 4. $t_2 = c * a$

a_0, b_0, c_0, d_0, e_3 5 $t_3 = t_2 - e$

a_0, b_0, c_6, d_0, e_3 6. $c = t_3$

7. $f = e + d$

$a_0, b_0, c_6, d_0, e_3, f_7$

$a_0, b_0, c_6, d_0, e_3, f_7$

$a_0, b_0, c_6, d_0, e_3, f_7$ 8. $t_4 = e * c$

$a_0, b_0, c_6, d_0, e_3, f_7$ 9. $t_5 = t_4 - f$

$a_0, b_0, c_6, d_{10}, e_3, f_7$ 10. $d = t_5$

11. $a = c + d$ $a_0, b_0, c_6, d_0, e_3, f_7$

12. $t_6 = b * a$ $a_{11}, b_0, c_6, d_0, e_3, f_7$

13. $t_7 = e - t_6$ $a_{11}, b_0, c_6, d_0, e_3, f_7$

14. $f = t_7$ $a_{11}, b_0, c_6, d_0, e_3, f_7$

$a_{11}, b_0, c_6, d_0, e_3, f_{14}$

$a_0, a_{11}, b_0, c_6, d_0, d_{10}, e_3, f_7, f_{14}$

$a_{15}, b_0, c_6, d_0, d_{10}, e_3, f_7, f_{14}$ 15. $a = c * b$

$a_{15}, b_0, c_6, d_0, d_{10}, e_3, f_7, f_{14}$ 16. $t_8 = f + d$

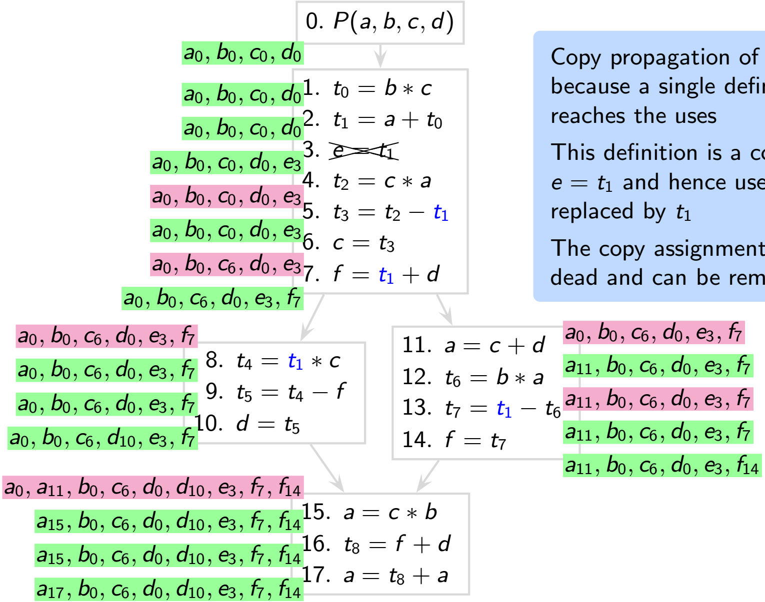
$a_{17}, b_0, c_6, d_0, d_{10}, e_3, f_7, f_{14}$ 17. $a = t_8 + a$

Copy propagation of e is possible because a single definition of e (e_3) reaches the uses

This definition is a copy assignment $e = t_1$ and hence uses of e can be replaced by t_1

The copy assignment $e = t_1$ becomes dead and can be removed

Copy Propagation Optimization in Our Program



0. $P(a, b, c, d)$

a_0, b_0, c_0, d_0

a_0, b_0, c_0, d_0

1. $t_0 = b * c$

a_0, b_0, c_0, d_0

2. $t_1 = a + t_0$

a_0, b_0, c_0, d_0, e_3

3. $\cancel{e = t_1}$

a_0, b_0, c_0, d_0, e_3

4. $t_2 = c * a$

a_0, b_0, c_0, d_0, e_3

5. $t_3 = t_2 - t_1$

a_0, b_0, c_0, d_0, e_3

6. $c = t_3$

a_0, b_0, c_6, d_0, e_3

7. $f = t_1 + d$

$a_0, b_0, c_6, d_0, e_3, f_7$

$a_0, b_0, c_6, d_0, e_3, f_7$

8. $t_4 = t_1 * c$

$a_0, b_0, c_6, d_0, e_3, f_7$

9. $t_5 = t_4 - f$

$a_0, b_0, c_6, d_0, e_3, f_7$

10. $d = t_5$

$a_0, b_0, c_6, d_{10}, e_3, f_7$

$a_0, b_0, c_6, d_0, e_3, f_7$

11. $a = c + d$

$a_{11}, b_0, c_6, d_0, e_3, f_7$

12. $t_6 = b * a$

$a_{11}, b_0, c_6, d_0, e_3, f_7$

13. $t_7 = t_1 - t_6$

$a_{11}, b_0, c_6, d_0, e_3, f_7$

14. $f = t_7$

$a_{11}, b_0, c_6, d_0, e_3, f_{14}$

$a_0, a_{11}, b_0, c_6, d_0, d_{10}, e_3, f_7, f_{14}$

15. $a = c * b$

$a_{15}, b_0, c_6, d_0, d_{10}, e_3, f_7, f_{14}$

16. $t_8 = f + d$

$a_{15}, b_0, c_6, d_0, d_{10}, e_3, f_7, f_{14}$

17. $a = t_8 + a$

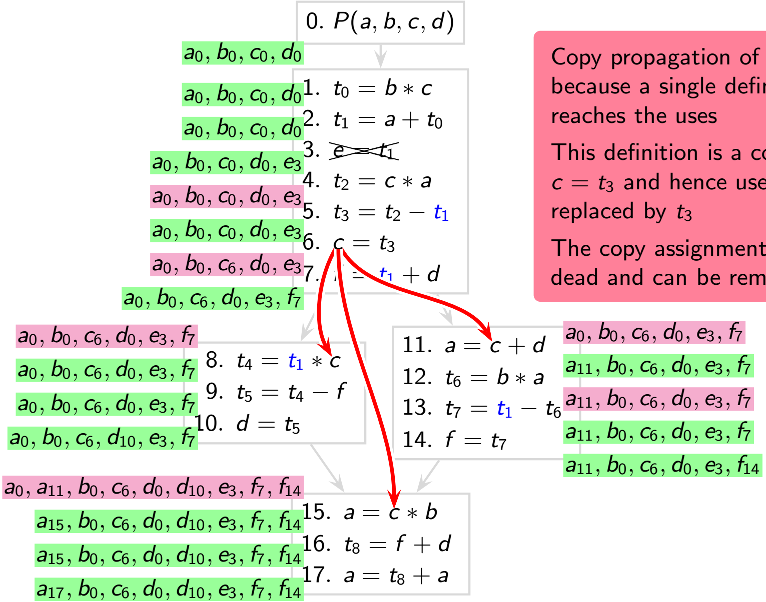
$a_{17}, b_0, c_6, d_0, d_{10}, e_3, f_7, f_{14}$

Copy propagation of e is possible because a single definition of e (e_3) reaches the uses

This definition is a copy assignment $e = t_1$ and hence uses of e can be replaced by t_1

The copy assignment $e = t_1$ becomes dead and can be removed

Copy Propagation Optimization in Our Program



0. $P(a, b, c, d)$

a_0, b_0, c_0, d_0

a_0, b_0, c_0, d_0
a_0, b_0, c_0, d_0
a_0, b_0, c_0, d_0, e_3
a_0, b_0, c_0, d_0, e_3
a_0, b_0, c_0, d_0, e_3
a_0, b_0, c_6, d_0, e_3
$a_0, b_0, c_6, d_0, e_3, f_7$

1. $t_0 = b * c$
2. $t_1 = a + t_0$
3. ~~$e = t_1$~~
4. $t_2 = c * a$
5. $t_3 = t_2 - t_1$
6. $c = t_3$
7. $f = t_1 + d$

$a_0, b_0, c_6, d_0, e_3, f_7$
$a_0, b_0, c_6, d_0, e_3, f_7$
$a_0, b_0, c_6, d_0, e_3, f_7$
$a_0, b_0, c_6, d_{10}, e_3, f_7$

8. $t_4 = t_1 * c$
9. $t_5 = t_4 - f$
10. $d = t_5$

11. $a = c + d$
12. $t_6 = b * a$
13. $t_7 = t_1 - t_6$
14. $f = t_7$

$a_0, b_0, c_6, d_0, e_3, f_7$
$a_{11}, b_0, c_6, d_0, e_3, f_7$
$a_{11}, b_0, c_6, d_0, e_3, f_7$
$a_{11}, b_0, c_6, d_0, e_3, f_7$
$a_{11}, b_0, c_6, d_0, e_3, f_{14}$

$a_0, a_{11}, b_0, c_6, d_0, d_{10}, e_3, f_7, f_{14}$
$a_{15}, b_0, c_6, d_0, d_{10}, e_3, f_7, f_{14}$
$a_{15}, b_0, c_6, d_0, d_{10}, e_3, f_7, f_{14}$
$a_{17}, b_0, c_6, d_0, d_{10}, e_3, f_7, f_{14}$

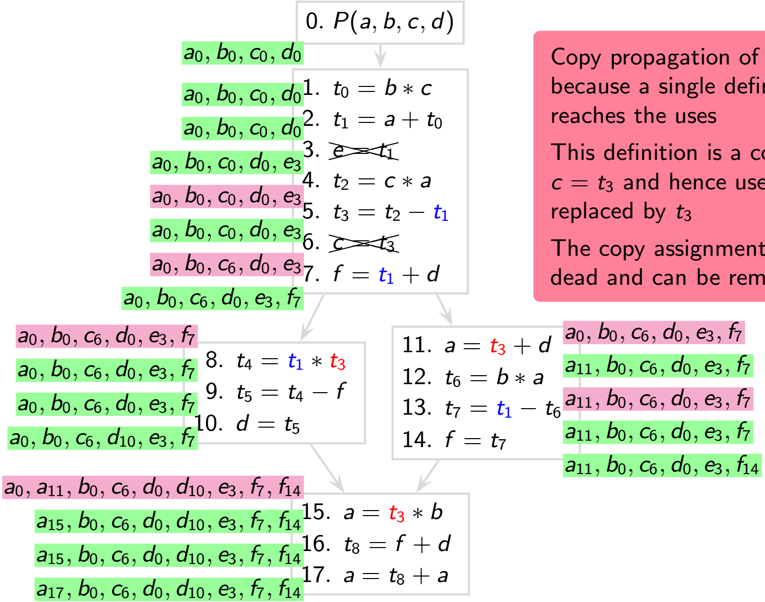
15. $a = c * b$
16. $t_8 = f + d$
17. $a = t_8 + a$

Copy propagation of c is possible because a single definition of c (c_6) reaches the uses

This definition is a copy assignment $c = t_3$ and hence uses of c can be replaced by t_3

The copy assignment $c = t_3$ becomes dead and can be removed

Copy Propagation Optimization in Our Program



0. $P(a, b, c, d)$

a_0, b_0, c_0, d_0

a_0, b_0, c_0, d_0
1. $t_0 = b * c$

a_0, b_0, c_0, d_0
2. $t_1 = a + t_0$

a_0, b_0, c_0, d_0, e_3
3. ~~$e = t_1$~~

a_0, b_0, c_0, d_0, e_3
4. $t_2 = c * a$

a_0, b_0, c_0, d_0, e_3
5. $t_3 = t_2 - t_1$

a_0, b_0, c_0, d_0, e_3
6. ~~$c = t_3$~~

a_0, b_0, c_6, d_0, e_3
7. $f = t_1 + d$

$a_0, b_0, c_6, d_0, e_3, f_7$

$a_0, b_0, c_6, d_0, e_3, f_7$
8. $t_4 = t_1 * t_3$

$a_0, b_0, c_6, d_0, e_3, f_7$
9. $t_5 = t_4 - f$

$a_0, b_0, c_6, d_0, e_3, f_7$
10. $d = t_5$

$a_0, b_0, c_6, d_{10}, e_3, f_7$

11. $a = t_3 + d$
$a_0, b_0, c_6, d_0, e_3, f_7$

12. $t_6 = b * a$
$a_{11}, b_0, c_6, d_0, e_3, f_7$

13. $t_7 = t_1 - t_6$
$a_{11}, b_0, c_6, d_0, e_3, f_7$

14. $f = t_7$
$a_{11}, b_0, c_6, d_0, e_3, f_7$

$a_{11}, b_0, c_6, d_0, e_3, f_{14}$

$a_0, a_{11}, b_0, c_6, d_0, d_{10}, e_3, f_7, f_{14}$
15. $a = t_3 * b$

$a_{15}, b_0, c_6, d_0, d_{10}, e_3, f_7, f_{14}$
16. $t_8 = f + d$

$a_{15}, b_0, c_6, d_0, d_{10}, e_3, f_7, f_{14}$
17. $a = t_8 + a$

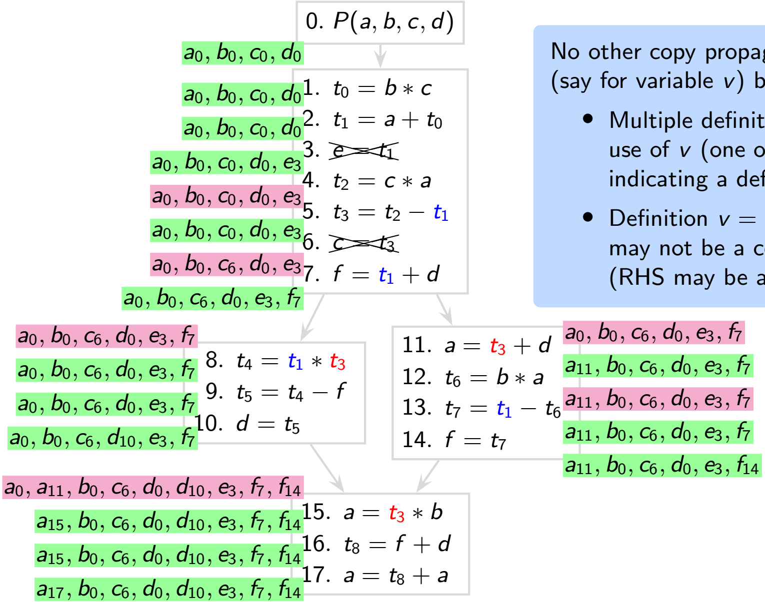
$a_{17}, b_0, c_6, d_0, d_{10}, e_3, f_7, f_{14}$

Copy propagation of c is possible because a single definition of c (c_6) reaches the uses

This definition is a copy assignment $c = t_3$ and hence uses of c can be replaced by t_3

The copy assignment $c = t_3$ becomes dead and can be removed

Copy Propagation Optimization in Our Program



0. $P(a, b, c, d)$

a_0, b_0, c_0, d_0

a_0, b_0, c_0, d_0 1. $t_0 = b * c$

a_0, b_0, c_0, d_0 2. $t_1 = a + t_0$

a_0, b_0, c_0, d_0, e_3 3. $\xcancel{e = t_1}$

a_0, b_0, c_0, d_0, e_3 4. $t_2 = c * a$

a_0, b_0, c_0, d_0, e_3 5. $t_3 = t_2 - t_1$

a_0, b_0, c_0, d_0, e_3 6. $\xcancel{c = t_3}$

a_0, b_0, c_6, d_0, e_3 7. $f = t_1 + d$

$a_0, b_0, c_6, d_0, e_3, f_7$

$a_0, b_0, c_6, d_0, e_3, f_7$

$a_0, b_0, c_6, d_0, e_3, f_7$ 8. $t_4 = t_1 * t_3$

$a_0, b_0, c_6, d_0, e_3, f_7$ 9. $t_5 = t_4 - f$

$a_0, b_0, c_6, d_{10}, e_3, f_7$ 10. $d = t_5$

11. $a = t_3 + d$ $a_0, b_0, c_6, d_0, e_3, f_7$

12. $t_6 = b * a$ $a_{11}, b_0, c_6, d_0, e_3, f_7$

13. $t_7 = t_1 - t_6$ $a_{11}, b_0, c_6, d_0, e_3, f_7$

14. $f = t_7$ $a_{11}, b_0, c_6, d_0, e_3, f_7$

$a_{11}, b_0, c_6, d_0, e_3, f_{14}$

$a_0, a_{11}, b_0, c_6, d_0, d_{10}, e_3, f_7, f_{14}$

$a_{15}, b_0, c_6, d_0, d_{10}, e_3, f_7, f_{14}$ 15. $a = t_3 * b$

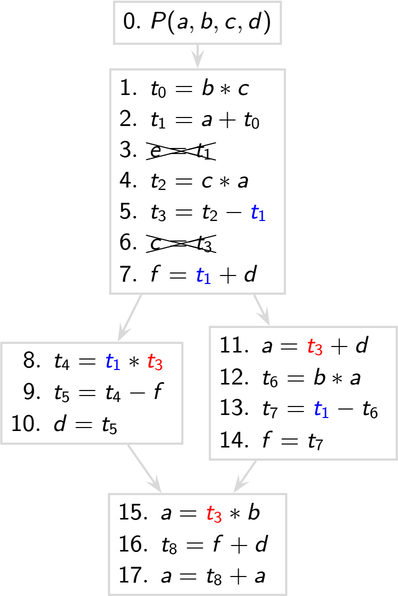
$a_{15}, b_0, c_6, d_0, d_{10}, e_3, f_7, f_{14}$ 16. $t_8 = f + d$

$a_{17}, b_0, c_6, d_0, d_{10}, e_3, f_7, f_{14}$ 17. $a = t_8 + a$

No other copy propagation can be done (say for variable v) because

- Multiple definitions of v reach a use of v (one of them could be v_0 indicating a definition-free path)
- Definition $v = \ldots$ of variable v may not be a copy assignment (RHS may be an expression)

Copy Propagation Optimization in Our Program



0. $P(a, b, c, d)$

1. $t_0 = b * c$
2. $t_1 = a + t_0$
3. ~~$e = t_1$~~
4. $t_2 = c * a$
5. $t_3 = t_2 - t_1$
6. ~~$c = t_3$~~
7. $f = t_1 + d$

8. $t_4 = t_1 * t_3$
9. $t_5 = t_4 - f$
10. $d = t_5$

11. $a = t_3 + d$
12. $t_6 = b * a$
13. $t_7 = t_1 - t_6$
14. $f = t_7$

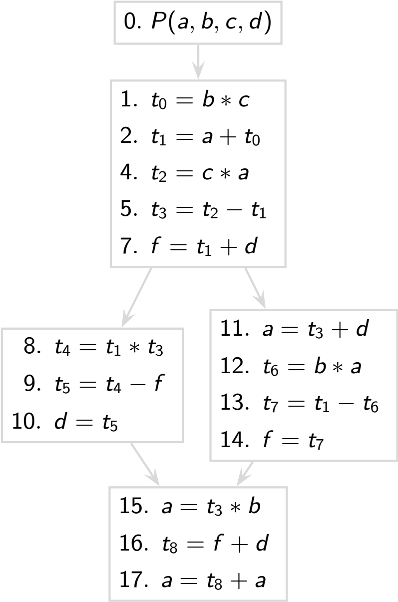
15. $a = t_3 * b$
16. $t_8 = f + d$
17. $a = t_8 + a$

Discovering Live Ranges

- Perform live variables analysis
    - The use of a variable v in statement m makes it live before statement m
    - The definition of a variable v in statement m kills its liveness

  If a variable v is live after statement m, it suggests the existence of a control flow path from statement m to EXIT along which x is used before being modified
- Set up the data flow equations over the control flow graph and compute the least fixed point solution
- Statement n is in the live range of variable v if,
    - a definition of v reaches statement n and v is live just before of n, or
    - statement n defines v and v is live just after n

Discovering Live Ranges in Our Program



0. $P(a, b, c, d)$

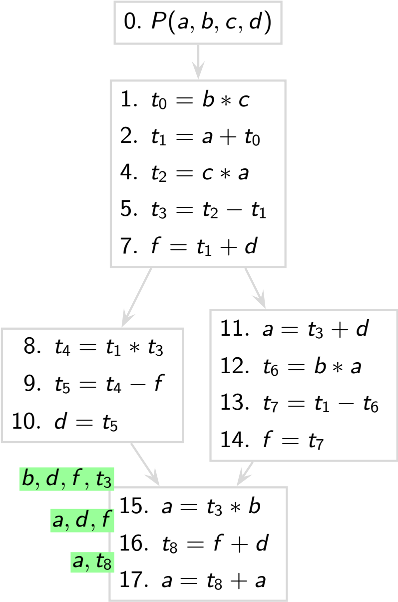
1. $t_0 = b * c$
2. $t_1 = a + t_0$
4. $t_2 = c * a$
5. $t_3 = t_2 - t_1$
7. $f = t_1 + d$

8. $t_4 = t_1 * t_3$
9. $t_5 = t_4 - f$
10. $d = t_5$

11. $a = t_3 + d$
12. $t_6 = b * a$
13. $t_7 = t_1 - t_6$
14. $f = t_7$

15. $a = t_3 * b$
16. $t_8 = f + d$
17. $a = t_8 + a$

Discovering Live Ranges in Our Program



0. $P(a, b, c, d)$

1. $t_0 = b * c$
2. $t_1 = a + t_0$
4. $t_2 = c * a$
5. $t_3 = t_2 - t_1$
7. $f = t_1 + d$

8. $t_4 = t_1 * t_3$
9. $t_5 = t_4 - f$
10. $d = t_5$

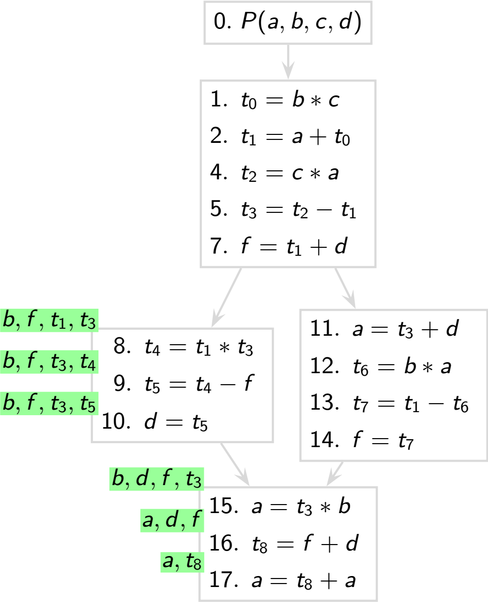
11. $a = t_3 + d$
12. $t_6 = b * a$
13. $t_7 = t_1 - t_6$
14. $f = t_7$

b, d, f, t_3 15. $a = t_3 * b$

a, d, f 16. $t_8 = f + d$

a, t_8 17. $a = t_8 + a$

Discovering Live Ranges in Our Program



0. $P(a, b, c, d)$

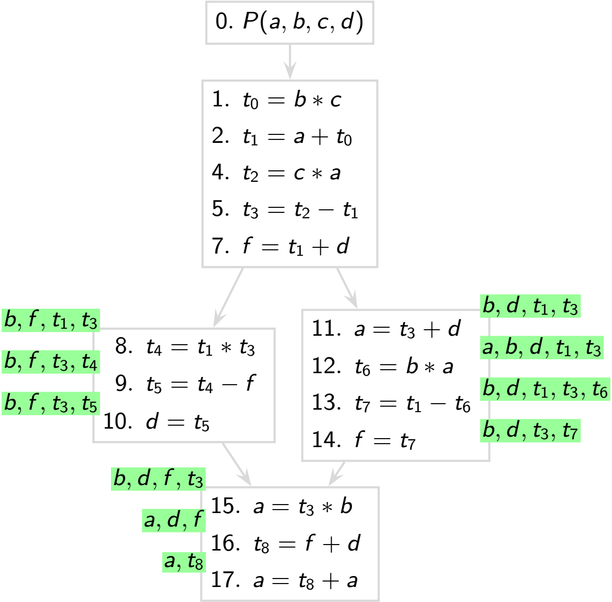
1. $t_0 = b * c$
2. $t_1 = a + t_0$
4. $t_2 = c * a$
5. $t_3 = t_2 - t_1$
7. $f = t_1 + d$

b, f, t_1, t_3
8. $t_4 = t_1 * t_3$
b, f, t_3, t_4
9. $t_5 = t_4 - f$
b, f, t_3, t_5
10. $d = t_5$

11. $a = t_3 + d$
12. $t_6 = b * a$
13. $t_7 = t_1 - t_6$
14. $f = t_7$

b, d, f, t_3
15. $a = t_3 * b$
a, d, f
16. $t_8 = f + d$
a, t_8
17. $a = t_8 + a$

Discovering Live Ranges in Our Program



0. $P(a, b, c, d)$

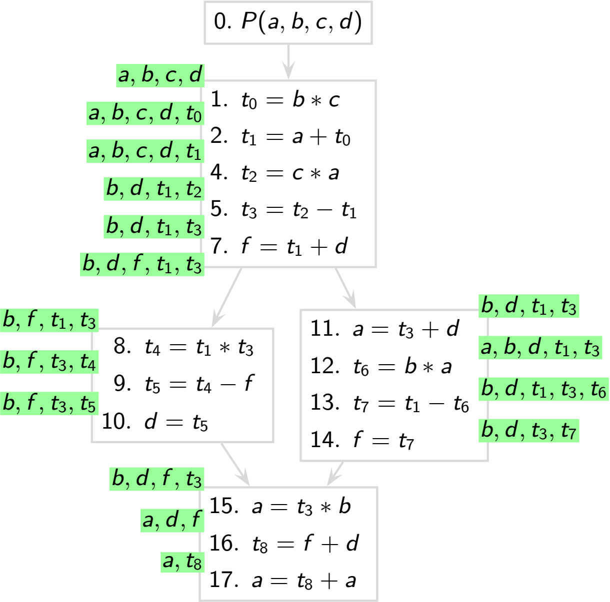
1. $t_0 = b * c$
2. $t_1 = a + t_0$
4. $t_2 = c * a$
5. $t_3 = t_2 - t_1$
7. $f = t_1 + d$

b, f, t_1, t_3
8. $t_4 = t_1 * t_3$
b, f, t_3, t_4
9. $t_5 = t_4 - f$
b, f, t_3, t_5
10. $d = t_5$

b, d, t_1, t_3
11. $a = t_3 + d$
a, b, d, t_1, t_3
12. $t_6 = b * a$
b, d, t_1, t_3, t_6
13. $t_7 = t_1 - t_6$
b, d, t_3, t_7
14. $f = t_7$

b, d, f, t_3
15. $a = t_3 * b$
a, d, f
16. $t_8 = f + d$
a, t_8
17. $a = t_8 + a$

Discovering Live Ranges in Our Program



0. $P(a, b, c, d)$

a, b, c, d
1. $t_0 = b * c$

a, b, c, d, t_0
2. $t_1 = a + t_0$

a, b, c, d, t_1
4. $t_2 = c * a$

b, d, t_1, t_2
5. $t_3 = t_2 - t_1$

b, d, t_1, t_3
7. $f = t_1 + d$

b, d, f, t_1, t_3

b, f, t_1, t_3
8. $t_4 = t_1 * t_3$

b, f, t_3, t_4
9. $t_5 = t_4 - f$

b, f, t_3, t_5
10. $d = t_5$

b, d, t_1, t_3
11. $a = t_3 + d$

a, b, d, t_1, t_3
12. $t_6 = b * a$

b, d, t_1, t_3, t_6
13. $t_7 = t_1 - t_6$

b, d, t_3, t_7
14. $f = t_7$

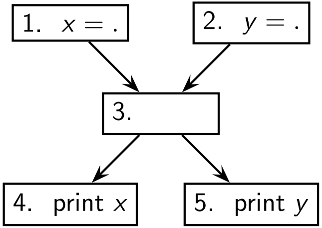
b, d, f, t_3
15. $a = t_3 * b$

a, d, f
16. $t_8 = f + d$

a, t_8
17. $a = t_8 + a$

Identifying Interference

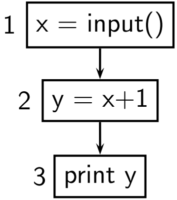
- Live ranges l_1 and l_2 interfere if a definition of the variable of l_1 occurs in l_2 or vice-versa
- Consider the example on the right
    - $l_x = \{1, 3, 4\}$ and $l_y = \{2, 3, 5\}$
    - $l_x \cap l_y \neq \emptyset$, yet the same register can be given to both x and y without any problem
    - l_x and l_y do not interfere



- Both x and y are live at the exit of nodes 1 and 2; however l_x does not include 2 (no definition of x at 2) and l_y does not include 1 (no definition of y at 1)
- Definitions of both x and y reach the entry of nodes 4 and 5; however l_x does not include 5 (x is not live in 5) and l_y does not include 4 (y is not live in 4)

Coalescing Beyond Copy Propagation

- Copy propagation eliminates copies only when a variable can be replaced by another variable or a constant

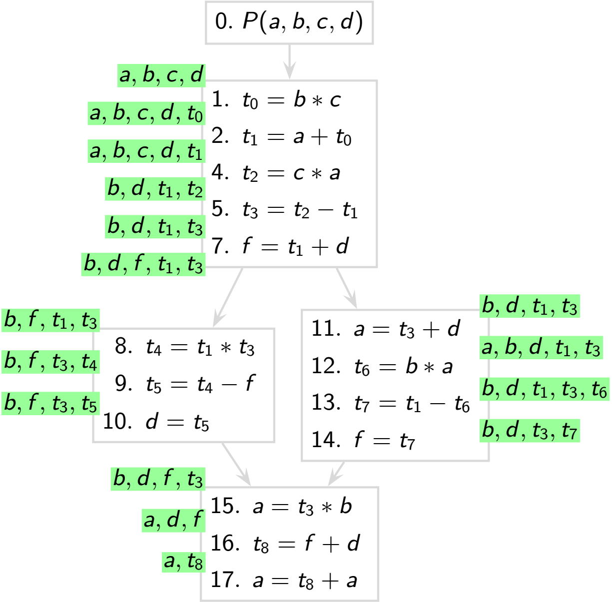


  - ○ No copy propagation possible
  - ○ $l_x = \{1, 2\}$, $l_y = \{2, 3\}$ and the definition of y occurs in a statement in l_x
  - ○ However, l_x and l_y can be coalesced because statement 2 is the last use of x and defines y

- If a statement defines a variable (say y) and the statement contains the "last use" of a definition of variable x (i.e., x is live at the entry of the statement but not at the exit of the statement)

  then the live ranges of x and y can be coalesced

Constructing the Interference Graph for Our Program



0. $P(a, b, c, d)$

a, b, c, d

1. $t_0 = b * c$

a, b, c, d, t_0

2. $t_1 = a + t_0$

a, b, c, d, t_1

4. $t_2 = c * a$

b, d, t_1, t_2

5. $t_3 = t_2 - t_1$

b, d, t_1, t_3

7. $f = t_1 + d$

b, d, f, t_1, t_3

b, f, t_1, t_3

8. $t_4 = t_1 * t_3$

b, f, t_3, t_4

9. $t_5 = t_4 - f$

b, f, t_3, t_5

10. $d = t_5$

b, d, t_1, t_3

11. $a = t_3 + d$

a, b, d, t_1, t_3

12. $t_6 = b * a$

b, d, t_1, t_3, t_6

13. $t_7 = t_1 - t_6$

b, d, t_3, t_7

14. $f = t_7$

b, d, f, t_3

15. $a = t_3 * b$

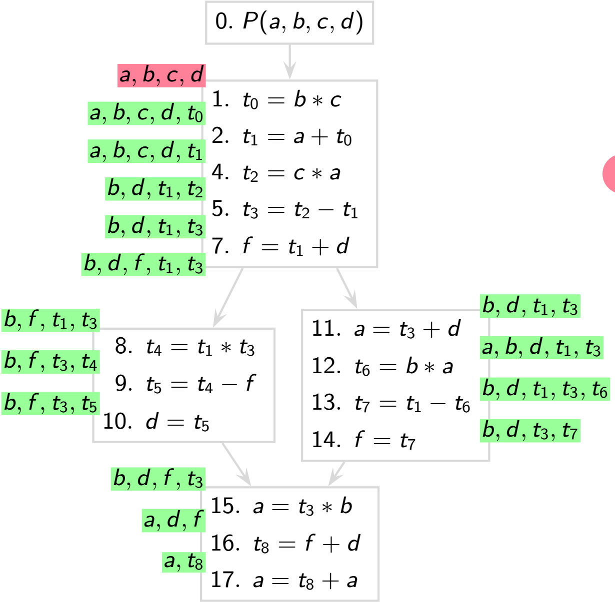
a, d, f

16. $t_8 = f + d$

a, t_8

17. $a = t_8 + a$

Constructing the Interference Graph for Our Program



0. $P(a, b, c, d)$

a, b, c, d

1. $t_0 = b * c$

a, b, c, d, t_0

2. $t_1 = a + t_0$

a, b, c, d, t_1

4. $t_2 = c * a$

b, d, t_1, t_2

5. $t_3 = t_2 - t_1$

b, d, t_1, t_3

7. $f = t_1 + d$

b, d, f, t_1, t_3

b, f, t_1, t_3

8. $t_4 = t_1 * t_3$

b, f, t_3, t_4

9. $t_5 = t_4 - f$

b, f, t_3, t_5

10. $d = t_5$

b, d, t_1, t_3

11. $a = t_3 + d$

a, b, d, t_1, t_3

12. $t_6 = b * a$

b, d, t_1, t_3, t_6

13. $t_7 = t_1 - t_6$

b, d, t_3, t_7

14. $f = t_7$

b, d, f, t_3

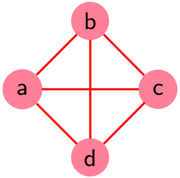
15. $a = t_3 * b$

a, d, f

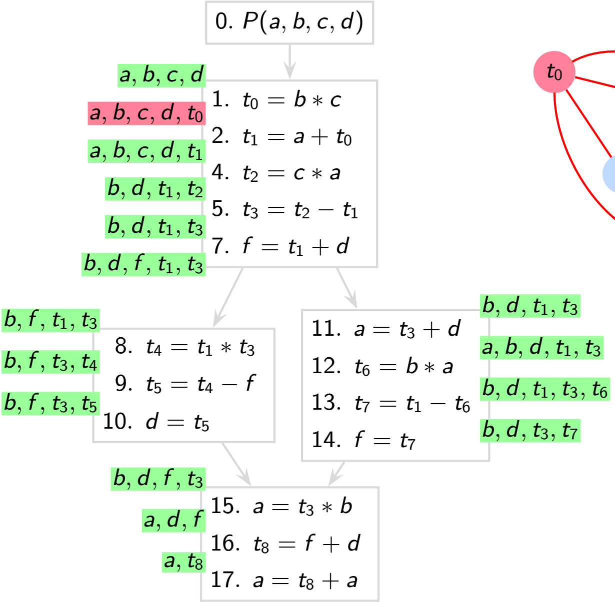
16. $t_8 = f + d$

a, t_8

17. $a = t_8 + a$



Constructing the Interference Graph for Our Program



0. $P(a, b, c, d)$

a, b, c, d
1. $t_0 = b * c$

a, b, c, d, t_0
2. $t_1 = a + t_0$

a, b, c, d, t_1
4. $t_2 = c * a$

b, d, t_1, t_2
5. $t_3 = t_2 - t_1$

b, d, t_1, t_3
7. $f = t_1 + d$

b, d, f, t_1, t_3

b, f, t_1, t_3
8. $t_4 = t_1 * t_3$

b, f, t_3, t_4
9. $t_5 = t_4 - f$

b, f, t_3, t_5
10. $d = t_5$

b, d, t_1, t_3
11. $a = t_3 + d$

a, b, d, t_1, t_3
12. $t_6 = b * a$

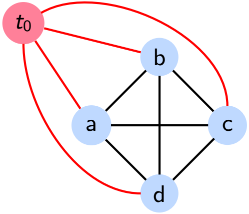
b, d, t_1, t_3, t_6
13. $t_7 = t_1 - t_6$

b, d, t_3, t_7
14. $f = t_7$

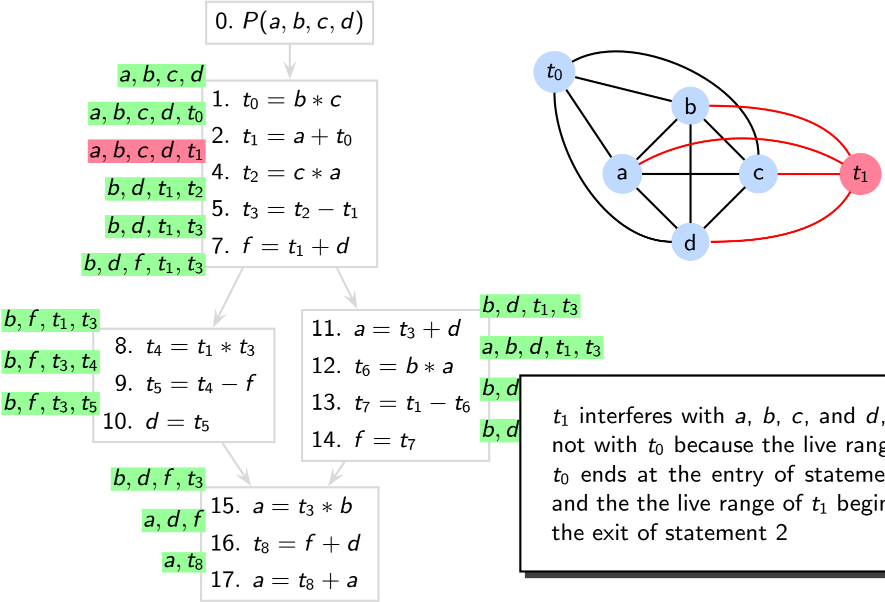
b, d, f, t_3
15. $a = t_3 * b$

a, d, f
16. $t_8 = f + d$

a, t_8
17. $a = t_8 + a$



Constructing the Interference Graph for Our Program



0. $P(a, b, c, d)$

a, b, c, d

1. $t_0 = b * c$

a, b, c, d, t_0

2. $t_1 = a + t_0$

a, b, c, d, t_1

4. $t_2 = c * a$

b, d, t_1, t_2

5. $t_3 = t_2 - t_1$

b, d, t_1, t_3

7. $f = t_1 + d$

b, d, f, t_1, t_3

b, f, t_1, t_3

8. $t_4 = t_1 * t_3$

b, f, t_3, t_4

9. $t_5 = t_4 - f$

b, f, t_3, t_5

10. $d = t_5$

b, d, t_1, t_3

11. $a = t_3 + d$

a, b, d, t_1, t_3

12. $t_6 = b * a$

b, d

13. $t_7 = t_1 - t_6$

b, d

14. $f = t_7$

b, d, f, t_3

15. $a = t_3 * b$

a, d, f

16. $t_8 = f + d$

a, t_8

17. $a = t_8 + a$

t_1 interferes with a, b, c, and d, but not with t_0 because the live range of t_0 ends at the entry of statement 2 and the the live range of t_1 begins at the exit of statement 2

Constructing the Interference Graph for Our Program

0. $P(a, b, c, d)$

a, b, c, d

1. $t_0 = b * c$

a, b, c, d, t_0

2. $t_1 = a + t_0$

a, b, c, d, t_1

4. $t_2 = c * a$

b, d, t_1, t_2

5. $t_3 = t_2 - t_1$

b, d, t_1, t_3

7. $f = t_1 + d$

b, d, f, t_1, t_3

b, f, t_1, t_3

8. $t_4 = t_1 * t_3$

b, f, t_3, t_4

9. $t_5 = t_4 - f$

b, f, t_3, t_5

10. $d = t_5$

b, d, t_1, t_3

11. $a = t_3 + d$

a, b, d, t_1, t_3

12. $t_6 = b * a$

b, d

13. $t_7 = t_1 - t_6$

b, d

14. $f = t_7$

b, d, f, t_3

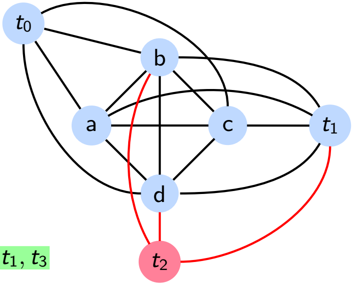
15. $a = t_3 * b$

a, d, f

16. $t_8 = f + d$

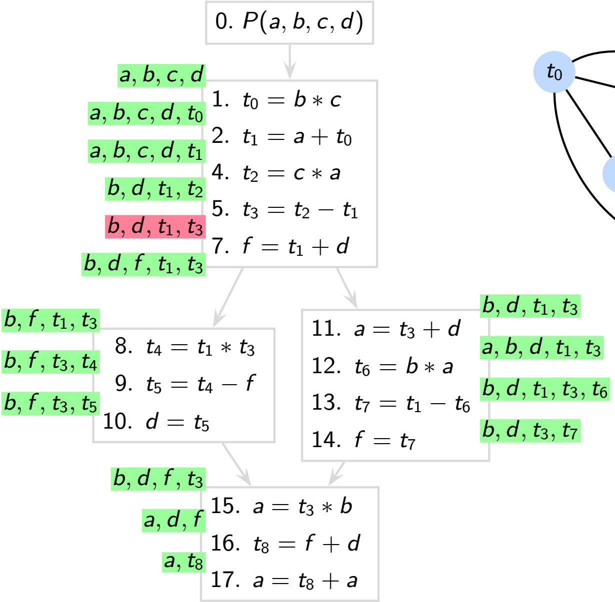
a, t_8

17. $a = t_8 + a$



t_2 interferes with b, d, and t_1, but not with a and c because the live range of a and c end at the entry of statement 3 and the the live range of t_2 begins at the exit of statement 3

Constructing the Interference Graph for Our Program



0. $P(a, b, c, d)$

a, b, c, d

1. $t_0 = b * c$

a, b, c, d, t_0

2. $t_1 = a + t_0$

a, b, c, d, t_1

4. $t_2 = c * a$

b, d, t_1, t_2

5. $t_3 = t_2 - t_1$

b, d, t_1, t_3

7. $f = t_1 + d$

b, d, f, t_1, t_3

b, f, t_1, t_3

8. $t_4 = t_1 * t_3$

b, f, t_3, t_4

9. $t_5 = t_4 - f$

b, f, t_3, t_5

10. $d = t_5$

b, d, t_1, t_3

11. $a = t_3 + d$

a, b, d, t_1, t_3

12. $t_6 = b * a$

b, d, t_1, t_3, t_6

13. $t_7 = t_1 - t_6$

b, d, t_3, t_7

14. $f = t_7$

b, d, f, t_3

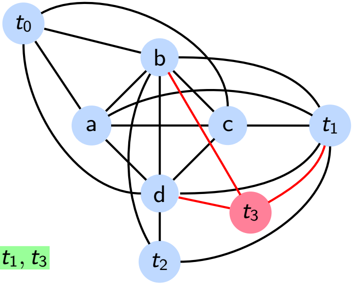
15. $a = t_3 * b$

a, d, f

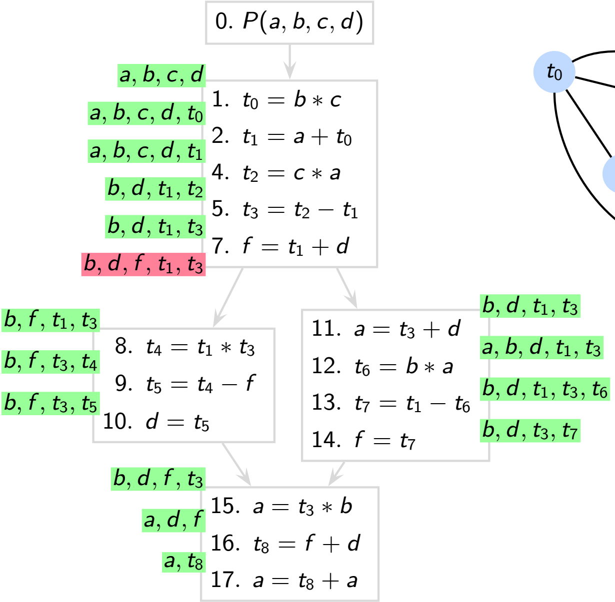
16. $t_8 = f + d$

a, t_8

17. $a = t_8 + a$



Constructing the Interference Graph for Our Program



0. $P(a, b, c, d)$

a, b, c, d

1. $t_0 = b * c$

a, b, c, d, t_0

2. $t_1 = a + t_0$

a, b, c, d, t_1

4. $t_2 = c * a$

b, d, t_1, t_2

5. $t_3 = t_2 - t_1$

b, d, t_1, t_3

7. $f = t_1 + d$

b, d, f, t_1, t_3

b, f, t_1, t_3

8. $t_4 = t_1 * t_3$

b, f, t_3, t_4

9. $t_5 = t_4 - f$

b, f, t_3, t_5

10. $d = t_5$

b, d, t_1, t_3

11. $a = t_3 + d$

a, b, d, t_1, t_3

12. $t_6 = b * a$

b, d, t_1, t_3, t_6

13. $t_7 = t_1 - t_6$

b, d, t_3, t_7

14. $f = t_7$

b, d, f, t_3

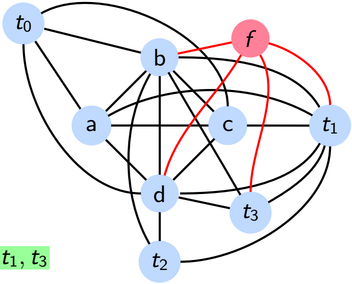
15. $a = t_3 * b$

a, d, f

16. $t_8 = f + d$

a, t_8

17. $a = t_8 + a$



Constructing the Interference Graph for Our Program

0. $P(a, b, c, d)$

a, b, c, d

1. $t_0 = b * c$

a, b, c, d, t_0

2. $t_1 = a + t_0$

a, b, c, d, t_1

4. $t_2 = c * a$

b, d, t_1, t_2

5. $t_3 = t_2 - t_1$

b, d, t_1, t_3

7. $f = t_1 + d$

b, d, f, t_1, t_3

b, f, t_1, t_3

8. $t_4 = t_1 * t_3$

b, f, t_3, t_4

9. $t_5 = t_4 - f$

b, f, t_3, t_5

10. $d = t_5$

b, d, t_1, t_3

11. $a = t_3 + d$

a, b, d, t_1, t_3

12. $t_6 = b * a$

b, d, t_1, t_3, t_6

13. $t_7 = t_1 - t_6$

b, d, t_3, t_7

14. $f = t_7$

b, d, f, t_3

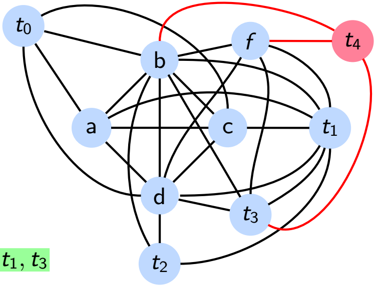
15. $a = t_3 * b$

a, d, f

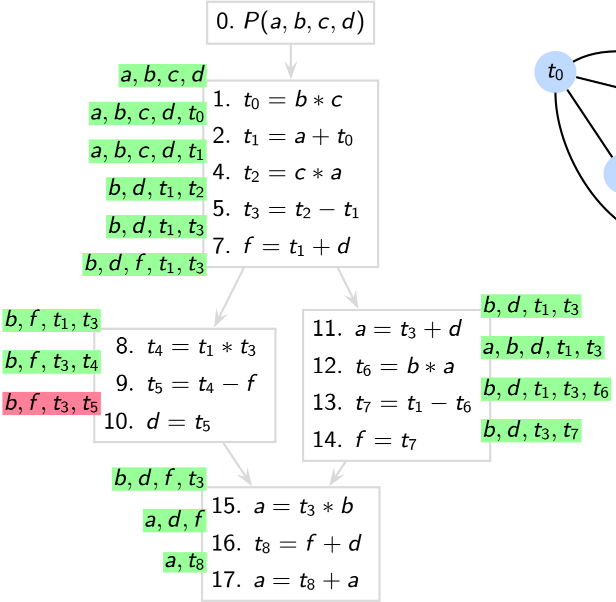
16. $t_8 = f + d$

a, t_8

17. $a = t_8 + a$



Constructing the Interference Graph for Our Program



0. $P(a, b, c, d)$

a, b, c, d
1. $t_0 = b * c$

a, b, c, d, t_0
2. $t_1 = a + t_0$

a, b, c, d, t_1
4. $t_2 = c * a$

b, d, t_1, t_2
5. $t_3 = t_2 - t_1$

b, d, t_1, t_3
7. $f = t_1 + d$

b, d, f, t_1, t_3

b, f, t_1, t_3
8. $t_4 = t_1 * t_3$

b, f, t_3, t_4
9. $t_5 = t_4 - f$

b, f, t_3, t_5
10. $d = t_5$

11. $a = t_3 + d$
b, d, t_1, t_3

12. $t_6 = b * a$
a, b, d, t_1, t_3

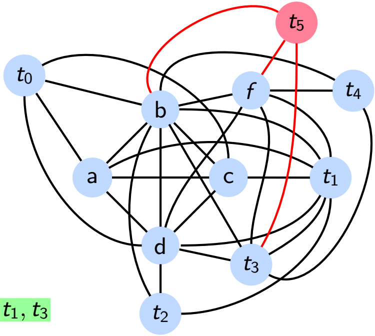
13. $t_7 = t_1 - t_6$
b, d, t_1, t_3, t_6

14. $f = t_7$
b, d, t_3, t_7

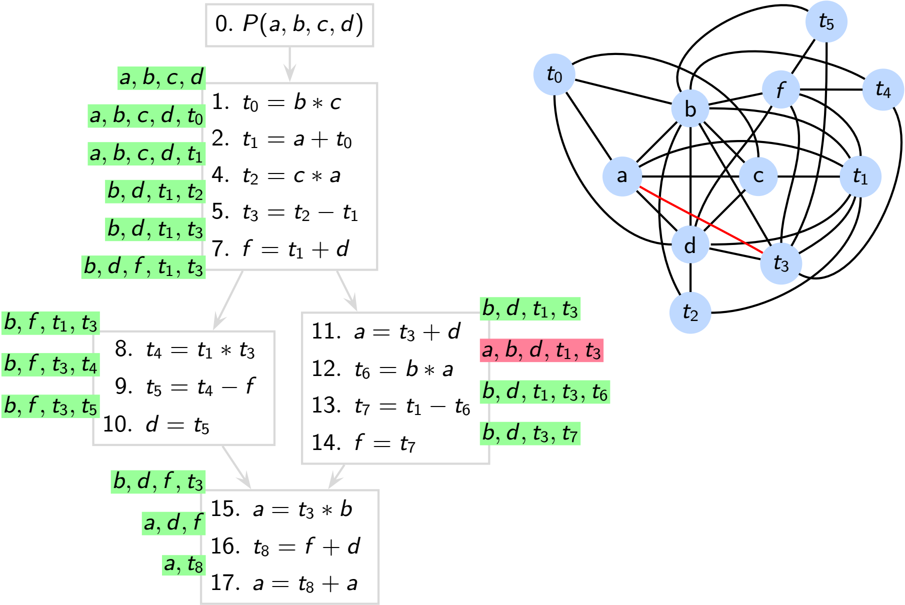
b, d, f, t_3
15. $a = t_3 * b$

a, d, f
16. $t_8 = f + d$

a, t_8
17. $a = t_8 + a$



Constructing the Interference Graph for Our Program



0. $P(a, b, c, d)$

a, b, c, d

1. $t_0 = b * c$

a, b, c, d, t_0

2. $t_1 = a + t_0$

a, b, c, d, t_1

4. $t_2 = c * a$

b, d, t_1, t_2

5. $t_3 = t_2 - t_1$

b, d, t_1, t_3

7. $f = t_1 + d$

b, d, f, t_1, t_3

b, f, t_1, t_3

8. $t_4 = t_1 * t_3$

b, f, t_3, t_4

9. $t_5 = t_4 - f$

b, f, t_3, t_5

10. $d = t_5$

b, d, t_1, t_3

11. $a = t_3 + d$

a, b, d, t_1, t_3

12. $t_6 = b * a$

b, d, t_1, t_3, t_6

13. $t_7 = t_1 - t_6$

b, d, t_3, t_7

14. $f = t_7$

b, d, f, t_3

15. $a = t_3 * b$

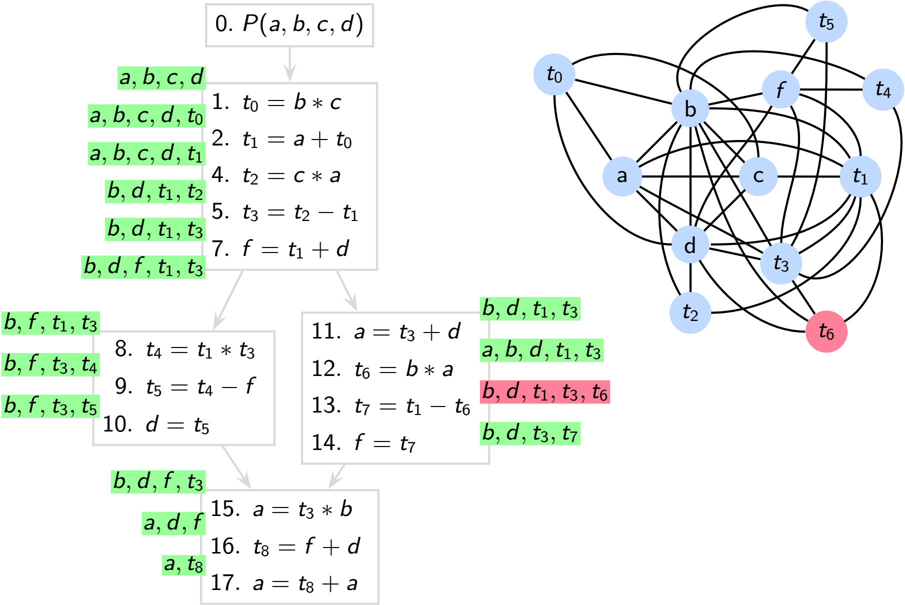
a, d, f

16. $t_8 = f + d$

a, t_8

17. $a = t_8 + a$

Constructing the Interference Graph for Our Program



0. $P(a, b, c, d)$

a, b, c, d

1. $t_0 = b * c$

a, b, c, d, t_0

2. $t_1 = a + t_0$

a, b, c, d, t_1

4. $t_2 = c * a$

b, d, t_1, t_2

5. $t_3 = t_2 - t_1$

b, d, t_1, t_3

7. $f = t_1 + d$

b, d, f, t_1, t_3

b, f, t_1, t_3

8. $t_4 = t_1 * t_3$

b, f, t_3, t_4

9. $t_5 = t_4 - f$

b, f, t_3, t_5

10. $d = t_5$

b, d, t_1, t_3

11. $a = t_3 + d$

a, b, d, t_1, t_3

12. $t_6 = b * a$

b, d, t_1, t_3, t_6

13. $t_7 = t_1 - t_6$

b, d, t_3, t_7

14. $f = t_7$

b, d, f, t_3

15. $a = t_3 * b$

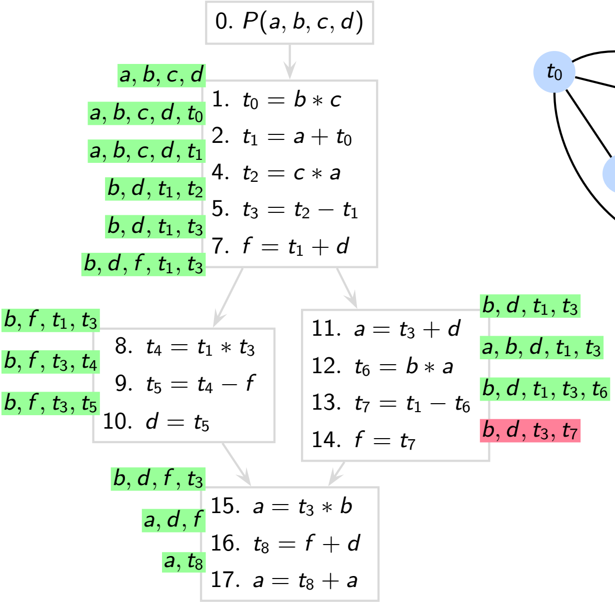
a, d, f

16. $t_8 = f + d$

a, t_8

17. $a = t_8 + a$

Constructing the Interference Graph for Our Program



0. $P(a, b, c, d)$

a, b, c, d

1. $t_0 = b * c$

a, b, c, d, t_0

2. $t_1 = a + t_0$

a, b, c, d, t_1

4. $t_2 = c * a$

b, d, t_1, t_2

5. $t_3 = t_2 - t_1$

b, d, t_1, t_3

7. $f = t_1 + d$

b, d, f, t_1, t_3

b, f, t_1, t_3

8. $t_4 = t_1 * t_3$

b, f, t_3, t_4

9. $t_5 = t_4 - f$

b, f, t_3, t_5

10. $d = t_5$

11. $a = t_3 + d$

b, d, t_1, t_3

12. $t_6 = b * a$

a, b, d, t_1, t_3

13. $t_7 = t_1 - t_6$

b, d, t_1, t_3, t_6

14. $f = t_7$

b, d, t_3, t_7

b, d, f, t_3

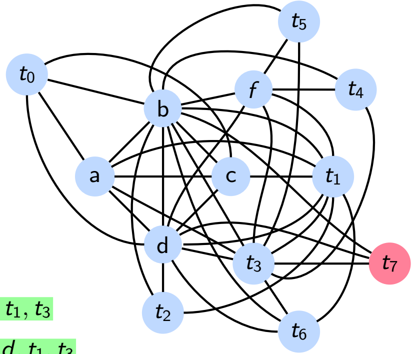
15. $a = t_3 * b$

a, d, f

16. $t_8 = f + d$

a, t_8

17. $a = t_8 + a$



Constructing the Interference Graph for Our Program

0. $P(a, b, c, d)$

a, b, c, d

1. $t_0 = b * c$

a, b, c, d, t_0

2. $t_1 = a + t_0$

a, b, c, d, t_1

4. $t_2 = c * a$

b, d, t_1, t_2

5. $t_3 = t_2 - t_1$

b, d, t_1, t_3

7. $f = t_1 + d$

b, d, f, t_1, t_3

b, f, t_1, t_3

8. $t_4 = t_1 * t_3$

b, f, t_3, t_4

9. $t_5 = t_4 - f$

b, f, t_3, t_5

10. $d = t_5$

b, d, t_1, t_3

11. $a = t_3 + d$

a, b, d, t_1, t_3

12. $t_6 = b * a$

b, d, t_1, t_3, t_6

13. $t_7 = t_1 - t_6$

b, d, t_3, t_7

14. $f = t_7$

b, d, f, t_3

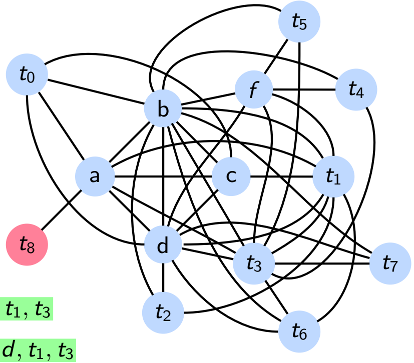
15. $a = t_3 * b$

a, d, f

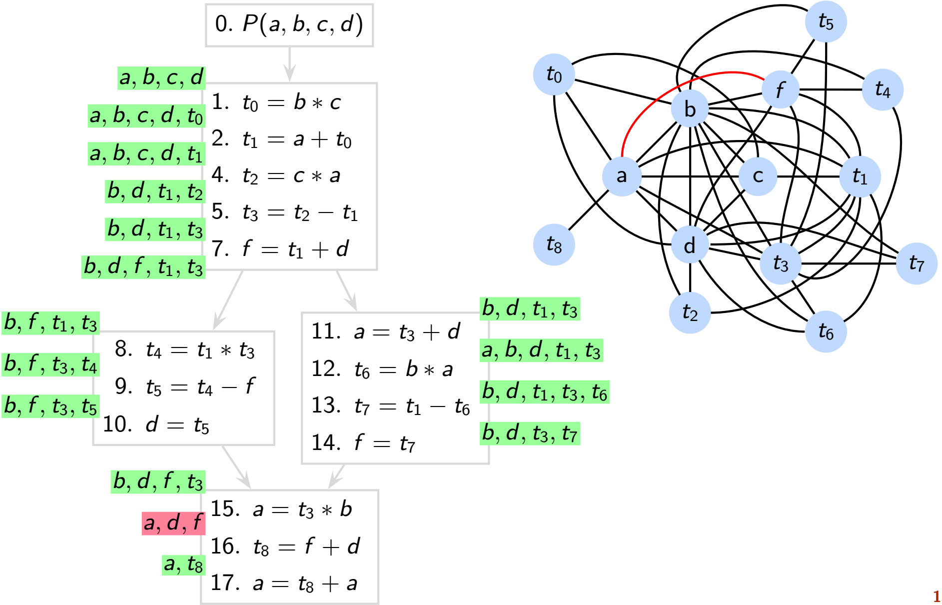
16. $t_8 = f + d$

a, t_8

17. $a = t_8 + a$



Constructing the Interference Graph for Our Program



0. $P(a, b, c, d)$

a, b, c, d
1. $t_0 = b * c$

a, b, c, d, t_0
2. $t_1 = a + t_0$

a, b, c, d, t_1
4. $t_2 = c * a$

b, d, t_1, t_2
5. $t_3 = t_2 - t_1$

b, d, t_1, t_3
7. $f = t_1 + d$

b, d, f, t_1, t_3

b, f, t_1, t_3
8. $t_4 = t_1 * t_3$

b, f, t_3, t_4
9. $t_5 = t_4 - f$

b, f, t_3, t_5
10. $d = t_5$

11. $a = t_3 + d$
b, d, t_1, t_3

12. $t_6 = b * a$
a, b, d, t_1, t_3

13. $t_7 = t_1 - t_6$
b, d, t_1, t_3, t_6

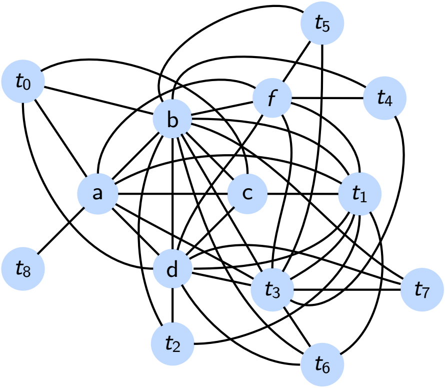
14. $f = t_7$
b, d, t_3, t_7

b, d, f, t_3
15. $a = t_3 * b$

a, d, f
16. $t_8 = f + d$

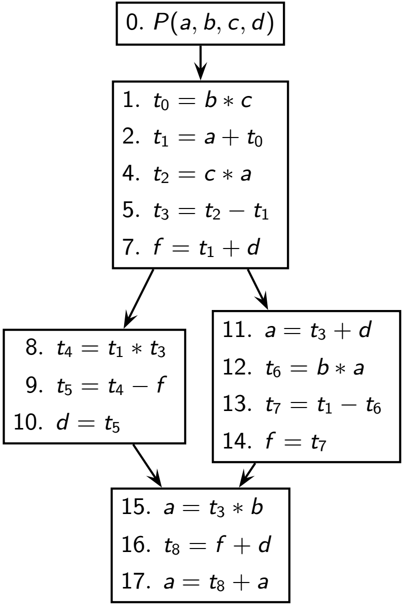
a, t_8
17. $a = t_8 + a$

The Resulting Interference Graph



Node	Degree
a	8
b	11
c	5
d	10
f	6
t_0	4
t_1	8
t_2	3
t_3	9
t_4	3
t_5	3
t_6	4
t_7	3
t_8	1

Live Range Information



0. $P(a, b, c, d)$

1. $t_0 = b * c$
2. $t_1 = a + t_0$
4. $t_2 = c * a$
5. $t_3 = t_2 - t_1$
7. $f = t_1 + d$

8. $t_4 = t_1 * t_3$
9. $t_5 = t_4 - f$
10. $d = t_5$

11. $a = t_3 + d$
12. $t_6 = b * a$
13. $t_7 = t_1 - t_6$
14. $f = t_7$

15. $a = t_3 * b$
16. $t_8 = f + d$
17. $a = t_8 + a$

Live range	Degree D	Loads L	Stores S	Spill cost $C = L + S$
a	8	4	3	7
b	11	3	0	3
c	5	2	0	2
d	10	3	1	4
f	6	2	2	4
t_0	4	1	1	2
t_1	8	4	1	5
t_2	3	1	1	2
t_3	9	3	1	4
t_4	3	1	1	2
t_5	3	1	1	2
t_6	4	1	1	2
t_7	3	1	1	2
t_8	1	1	1	2

Chaitin-Briggs Allocator

k-colouring using Chaitin's method

1. Simplify(a)

   Remove nodes in an arbitrary order s.t. for node n, $D(n) < k$

   Push them on a stack

k-colouring using Briggs' method

Since $D(n) < k$, we are guaranteed to find a colour for n

Chaitin-Briggs Allocator

k-colouring using Chaitin's method

1. Simplify(a)

   Remove nodes in an arbitrary order s.t. for node n, $D(n) < k$

   Push them on a stack

2. Simplify(b)

   If the graph is not empty, find the node with the least spill cost, spill it, and go back to step Simplify(a)

k-colouring using Briggs' method

Chaitin-Briggs Allocator

k-colouring using Chaitin's method

1. Simplify(a)

   Remove nodes in an arbitrary order s.t. for node n, $D(n) < k$

   Push them on a stack

2. Simplify(b)

   If the graph is not empty, find the node with the least spill cost, spill it, and go back to step Simplify(a)

3. Colour.

   Repeatedly pop the node from top of the stack, plug it in the graph and give it a colour distinct from its neighbours

k-colouring using Briggs' method

Chaitin-Briggs Allocator

k-colouring using Chaitin's method

1. Simplify(a)

   Remove nodes in an arbitrary order s.t. for node n, $D(n) < k$

   Push them on a stack

2. Simplify(b)

   If the graph is not empty, find the node with the least spill cost, spill it, and go back to step Simplify(a)

3. Colour.

   Repeatedly pop the node from top of the stack, plug it in the graph and give it a colour distinct from its neighbours

k-colouring using Briggs' method

1. Simplify(a)

   Remove nodes in an arbitrary order s.t. for node n, $D(n) < k$

   Push them on a stack

Chaitin-Briggs Allocator

k-colouring using Chaitin's method

1. Simplify(a)

   Remove nodes in an arbitrary order s.t. for node n, $D(n) < k$

   Push them on a stack

2. Simplify(b)

   If the graph is not empty, find the node with the least spill cost, spill it, and go back to step Simplify(a)

3. Colour.

   Repeatedly pop the node from top of the stack, plug it in the graph and give it a colour distinct from its neighbours

k-colouring using Briggs' method

1. Simplify(a)

   Remove nodes in an arbitrary order s.t. for node n, $D(n) < k$

   Push them on a stack

2. Simplify(b)

   If the graph is not empty, mark the nodes as potentially spillable and stack them

If $D(n) \geq k$, we may still find a colour for n if two neighbours of n do not interfere with each other and hence can get the same colour

Chaitin-Briggs Allocator

k-colouring using Chaitin's method

1. Simplify(a)

   Remove nodes in an arbitrary order s.t. for node n, $D(n) < k$

   Push them on a stack

2. Simplify(b)

   If the graph is
   the node with t
   cost, spill it, an
   step Simplify(a

3. Colour.

   Repeatedly pop the node from top of the stack, plug it in the graph and give it a colour distinct from its neighbours

k-colouring using Briggs' method

1. Simplify(a)

   Remove nodes in an arbitrary order s.t. for node n, $D(n) < k$

   Push them on a stack

   y, mark the
   lable and stack

   e from top of
   the stack, plug it in the graph and give it a colour distinct from its neighbours

   If a node cannot be coloured, spill it and go back to step Simplify(a)

- Chaitin's wisdom: Make the graph colourable and then colour it
- Briggs' wisdom: Colour the graph to find out if it is colourable

Chaitin-Briggs Allocator

k-colouring using Chaitin's method

1. Simplify(a)

   Remove nodes in an arbitrary order s.t. for node n, $D(n) < k$

   Push them on a stack

2. Simplify(b)

   If the graph is not empty, find the node with the least spill cost, spill it, and go back to step Simplify(a)

3. Colour.

   Repeatedly pop the node from top of the stack, plug it in the graph and give it a colour distinct from its neighbours

k-colouring using Briggs' method

1. Simplify(a)

   Remove nodes in an arbitrary order s.t. for node n, $D(n) < k$

   Push them on a stack

2. Simplify(b)

   If the graph is not empty, mark the nodes as potentially spillable and stack them

3. Colour.

   Repeatedly pop the node from top of the stack, plug it in the graph and give it a colour distinct from its neighbours

   If a node cannot be coloured, spill it and go back to step Simplify(a)

Spilling Decisions

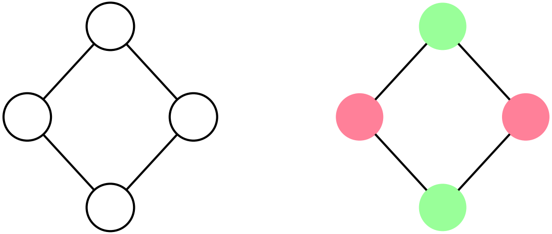
- Spill cost is weighted by loop nesting depth d

$$C(n) = (L(n) + S(n)) \times 10^d$$

- Sometime people normalize $C(n)$ by degree and consider the ratio $\frac{C(n)}{D(n)}$
- Spill decision should be taken for one live range at a time
  - When we conclude that we need spilling, we should spill a live range with the least cost and restart simplification after spilling the live range
    - in Simply(b) step in Chaitin's method
    - in Colour step in Briggs' modification
  - Spilling reduces the degree of other live ranges and another round of simplification may give us a better order of colouring the nodes

The Advantage of Delaying Spilling from Simplification to Colouring

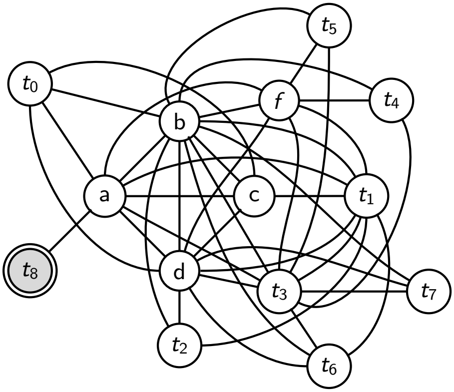
- Chaitin's method cannot colour the diamond graph with two colours but Brigg's method can



- Chaitin's method would spill live ranges because the degree of live ranges is not smaller than the number of colours
- Brigg's method would not spill before coloring and would find that the two neighbours of any node in this graph can be given the same colour

Simplify and Colour the Interference Graph (Briggs' Method)

5 Colours 🟨 🟦 🟫 🟩 🟥

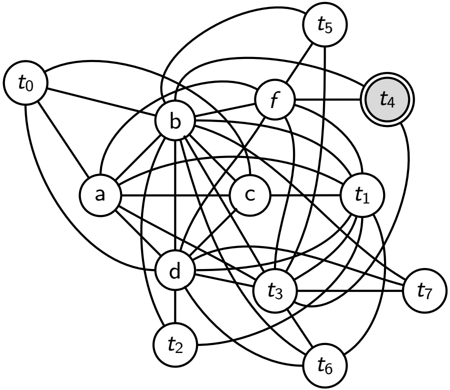


n	$D(n)$
t_8	< 5
t_4	
t_5	
t_7	
t_2	
t_0	
t_6	
c	≥ 5
a	
t_1	
t_3	
f	
d	
b	

Step Simplify(a)

Simplify and Colour the Interference Graph (Briggs' Method)

5 Colours 🟨 🟦 🟫 🟩 🟥



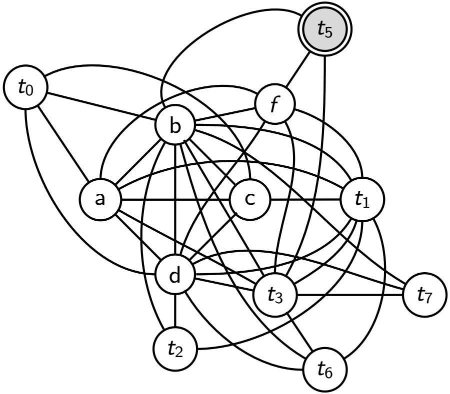
n	$D(n)$
t_8	< 5
t_4	
t_5	
t_7	
t_2	
t_0	
t_6	
c	≥ 5
a	
t_1	
t_3	
f	
d	
b	

Step Simplify(a)

t_8

Simplify and Colour the Interference Graph (Briggs' Method)

5 Colours 🟨 🟦 🟫 🟩 🟥



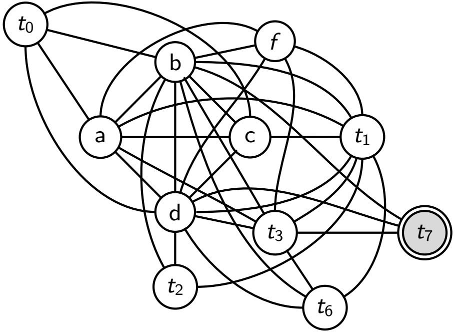
Step Simplify(a)

t_4
t_8

n	$D(n)$
t_8	< 5
t_4	
t_5	
t_7	
t_2	
t_0	
t_6	
c	≥ 5
a	
t_1	
t_3	
f	
d	
b	

Simplify and Colour the Interference Graph (Briggs' Method)

5 Colours 🟨 🟦 🟫 🟩 🟥



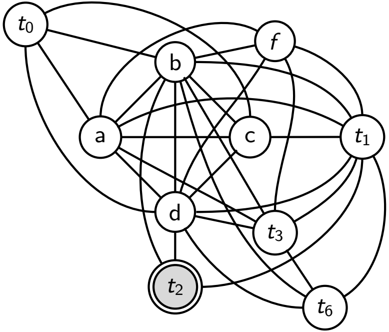
Step Simplify(a)

t_5
t_4
t_8

n	$D(n)$
t_8	< 5
t_4	
t_5	
t_7	
t_2	
t_0	
t_6	
c	≥ 5
a	
t_1	
t_3	
f	
d	
b	

Simplify and Colour the Interference Graph (Briggs' Method)

5 Colours 🟨 🟦 🟫 🟩 🟥



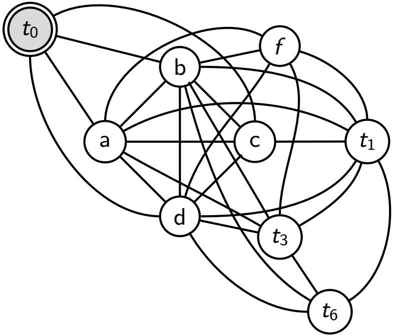
Step Simplify(a)

t_7
t_5
t_4
t_8

n	$D(n)$
t_8	< 5
t_4	
t_5	
t_7	
t_2	
t_0	
t_6	
c	≥ 5
a	
t_1	
t_3	
f	
d	
b	

Simplify and Colour the Interference Graph (Briggs' Method)

5 Colours 🟨 🟦 🟫 🟩 🟥



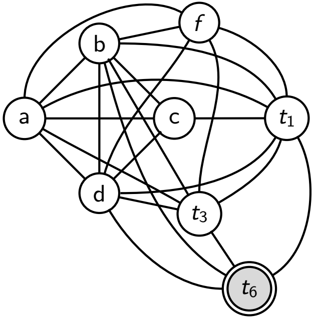
Step Simplify(a)

t_2
t_7
t_5
t_4
t_8

n	$D(n)$
t_8	< 5
t_4	
t_5	
t_7	
t_2	
t_0	
t_6	
c	≥ 5
a	
t_1	
t_3	
f	
d	
b	

Simplify and Colour the Interference Graph (Briggs' Method)

5 Colours 🟨 🟦 🟫 🟩 🟥



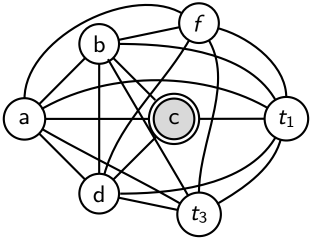
Step Simplify(a)

t_0
t_2
t_7
t_5
t_4
t_8

n	$D(n)$
t_8	< 5
t_4	
t_5	
t_7	
t_2	
t_0	
t_6	
c	≥ 5
a	
t_1	
t_3	
f	
d	
b	

Simplify and Colour the Interference Graph (Briggs' Method)

5 Colours 🟨 🟦 🟫 🟩 🟥



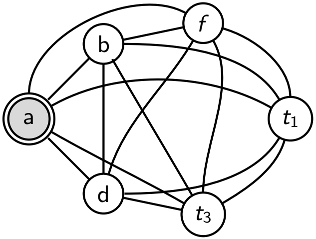
Step Simplify(a)

t_6
t_0
t_2
t_7
t_5
t_4
t_8

n	$D(n)$
t_8	< 5
t_4	
t_5	
t_7	
t_2	
t_0	
t_6	
c	≥ 5
a	
t_1	
t_3	
f	
d	
b	

Simplify and Colour the Interference Graph (Briggs' Method)

5 Colours 🟨 🟦 🟫 🟩 🟥



Step Simplify(b)

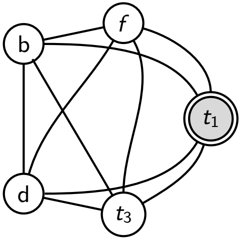
PS

c
t_6
t_0
t_2
t_7
t_5
t_4
t_8

n	$D(n)$
t_8	< 5
t_4	
t_5	
t_7	
t_2	
t_0	
t_6	
c	≥ 5
a	
t_1	
t_3	
f	
d	
b	

Simplify and Colour the Interference Graph (Briggs' Method)

5 Colours 🟨 🟦 🟫 🟩 🟥



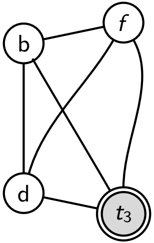
PS	a
PS	c
	t_6
	t_0
	t_2
	t_7
	t_5
	t_4
	t_8

n	$D(n)$
t_8 t_4 t_5 t_7 t_2 t_0 t_6	< 5
c a t_1 t_3 f d b	≥ 5

Step Simplify(b)

Simplify and Colour the Interference Graph (Briggs' Method)

5 Colours 🟨 🟦 🟫 🟩 🟥



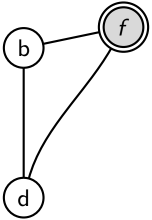
PS	t_1
PS	a
PS	c
	t_6
	t_0
	t_2
	t_7
	t_5
	t_4
	t_8

n	$D(n)$
t_8	< 5
t_4	
t_5	
t_7	
t_2	
t_0	
t_6	
c	≥ 5
a	
t_1	
t_3	
f	
d	
b	

Step Simplify(b)

Simplify and Colour the Interference Graph (Briggs' Method)

5 Colours 🟨 🟦 🟫 🟩 🟥



PS	t_3
PS	t_1
PS	a
PS	c
	t_6
	t_0
	t_2
	t_7
	t_5
	t_4
	t_8

n	$D(n)$
t_8	< 5
t_4	
t_5	
t_7	
t_2	
t_0	
t_6	
c	≥ 5
a	
t_1	
t_3	
f	
d	
b	

Step Simplify(b)

Simplify and Colour the Interference Graph (Briggs' Method)

5 Colours 🟨 🟦 🟫 🟩 🟥

PS	f
PS	t_3
PS	t_1
PS	a
PS	c
	t_6
	t_0
	t_2
	t_7
	t_5
	t_4
	t_8

n	$D(n)$
t_8	< 5
t_4	
t_5	
t_7	
t_2	
t_0	
t_6	
c	≥ 5
a	
t_1	
t_3	
f	
d	
b	

Step Simplify(b)

Simplify and Colour the Interference Graph (Briggs' Method)

5 Colours 🟨 🟦 🟫 🟩 🟥

PS	d
PS	f
PS	t_3
PS	t_1
PS	a
PS	c
	t_6
	t_0
	t_2
	t_7
	t_5
	t_4
	t_8

n	$D(n)$
t_8	< 5
t_4	
t_5	
t_7	
t_2	
t_0	
t_6	
c	≥ 5
a	
t_1	
t_3	
f	
d	
b	

Step Simplify(b)

Simplify and Colour the Interference Graph (Briggs' Method)

5 Colours 🟨 🟦 🟫 🟩 🟥

PS	b
PS	d
PS	f
PS	t_3
PS	t_1
PS	a
PS	c
	t_6
	t_0
	t_2
	t_7
	t_5
	t_4
	t_8

n	$D(n)$
t_8	< 5
t_4	
t_5	
t_7	
t_2	
t_0	
t_6	
c	≥ 5
a	
t_1	
t_3	
f	
d	
b	

Simplify and Colour the Interference Graph (Briggs' Method)

5 Colours 🟨 🟦 🟫 🟩 🟥

PS	b	🟨 🟦 🟫 🟩 🟥
PS	d	
PS	f	
PS	t_3	
PS	t_1	
PS	a	
PS	c	
	t_6	
	t_0	
	t_2	
	t_7	
	t_5	
	t_4	
	t_8	

Step Colour

Simplify and Colour the Interference Graph (Briggs' Method)

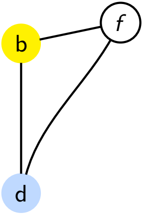
5 Colours 🟨 🟦 🟫 🟩 🟥

PS	d	🟦 🟫 🟩 🟥
PS	f	
PS	t_3	
PS	t_1	
PS	a	
PS	c	
	t_6	
	t_0	
	t_2	
	t_7	
	t_5	
	t_4	
	t_8	

Step Colour

Simplify and Colour the Interference Graph (Briggs' Method)

5 Colours 🟨 🟦 🟫 🟩 🟥

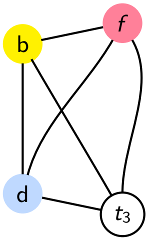


PS	f	🟫 🟩 🟥
PS	t_3	
PS	t_1	
PS	a	
PS	c	
	t_6	
	t_0	
	t_2	
	t_7	
	t_5	
	t_4	
	t_8	

Step Colour

Simplify and Colour the Interference Graph (Briggs' Method)

5 Colours 🟨 🟦 🟫 🟩 🟥

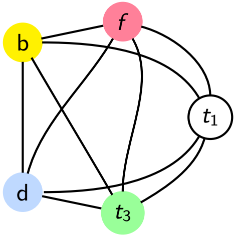


PS	t_3	🟩 🟫
PS	t_1	
PS	a	
PS	c	
	t_6	
	t_0	
	t_2	
	t_7	
	t_5	
	t_4	
	t_8	

Step Colour

Simplify and Colour the Interference Graph (Briggs' Method)

5 Colours 🟨 🟦 🟫 🟩 🟥



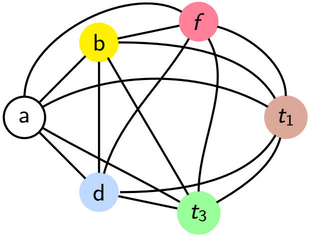
PS	t_1	🟫
PS	a	
PS	c	
	t_6	
	t_0	
	t_2	
	t_7	
	t_5	
	t_4	
	t_8	

Step Colour

Subgraph K_5 needs all 5 colours

Simplify and Colour the Interference Graph (Briggs' Method)

5 Colours 🟨 🟦 🟫 🟩 🟥



PS	a
PS	c
	t_6
	t_0
	t_2
	t_7
	t_5
	t_4
	t_8

No color (Subgraph K_6)

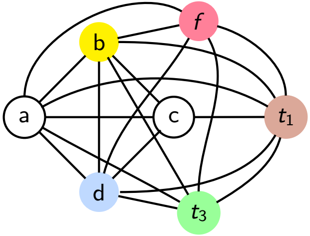
Need to spill a and restart simplification

Is not required for this example because the degree of c becomes 5

Step Colour

Simplify and Colour the Interference Graph (Briggs' Method)

5 Colours 🟨 🟦 🟫 🟩 🟥



PS

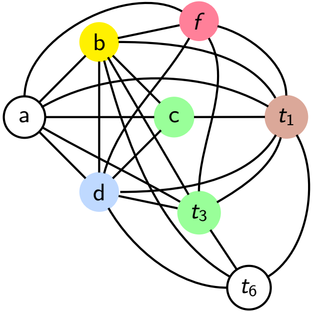
c
t_6
t_0
t_2
t_7
t_5
t_4
t_8

🟩 🟥

Step Colour

Simplify and Colour the Interference Graph (Briggs' Method)

5 Colours 🟨 🟦 🟫 🟩 🟥

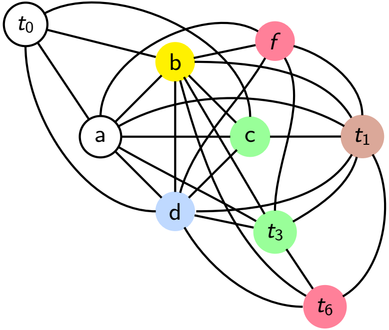


t_6 🟥
t_0
t_2
t_7
t_5
t_4
t_8

Step Colour

Simplify and Colour the Interference Graph (Briggs' Method)

5 Colours 🟨 🟦 🟫 🟩 🟥

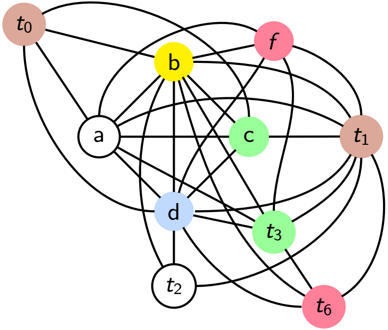


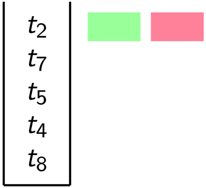
Step Colour

t_0 🟫 🟥
t_2
t_7
t_5
t_4
t_8

Simplify and Colour the Interference Graph (Briggs' Method)

5 Colours 🟨 🟦 🟫 🟩 🟥



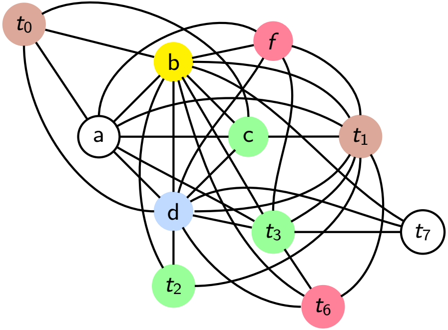


t_2 🟩 🟥
t_7
t_5
t_4
t_8

Step Colour

Simplify and Colour the Interference Graph (Briggs' Method)

5 Colours 🟨 🟦 🟫 🟩 🟥

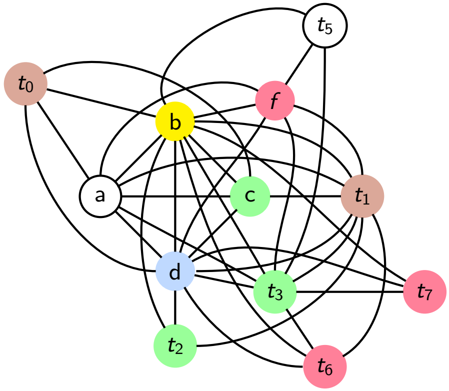


t_7 🟥 🟫
t_5
t_4
t_8

Step Colour

Simplify and Colour the Interference Graph (Briggs' Method)

5 Colours 🟨 🟦 🟫 🟩 🟥

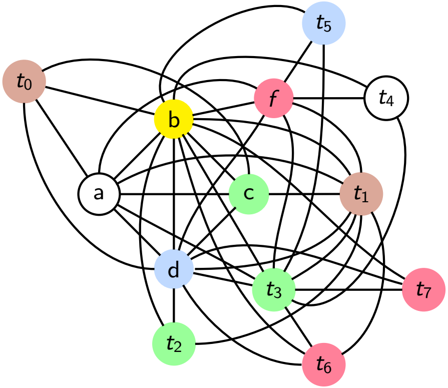


Step Colour

t_5 🟦 🟫
t_4
t_8

Simplify and Colour the Interference Graph (Briggs' Method)

5 Colours 🟨 🟦 🟫 🟩 🟥

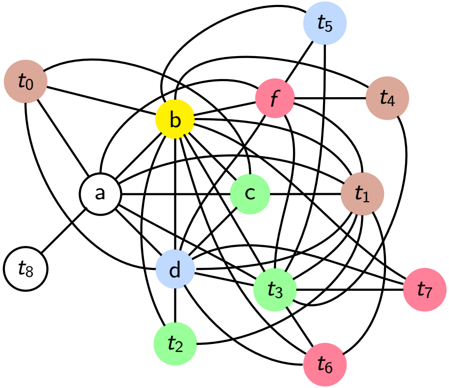


Step Colour

t_4 🟫 🟦
t_8

Simplify and Colour the Interference Graph (Briggs' Method)

5 Colours 🟨 🟦 🟫 🟩 🟥

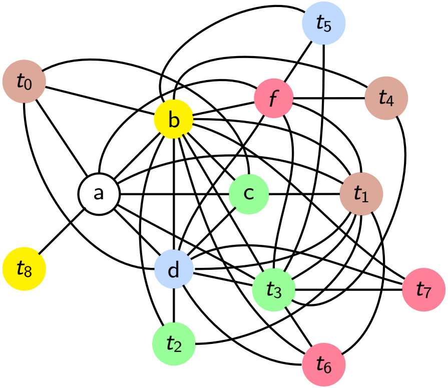


Step Colour

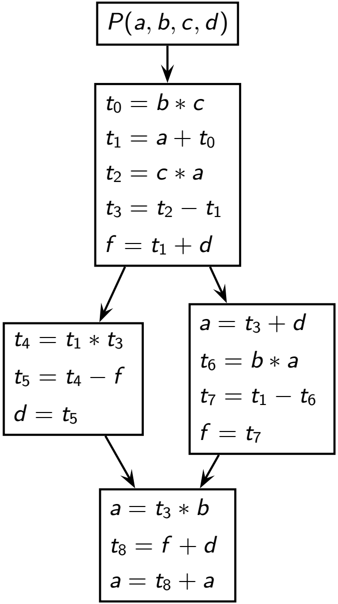
t_8 🟨 🟦 🟩 🟫

Simplify and Colour the Interference Graph (Briggs' Method)

5 Colours 🟨 🟦 🟫 🟩 🟥



Program After Global Register Allocation



$P(a, b, c, d)$

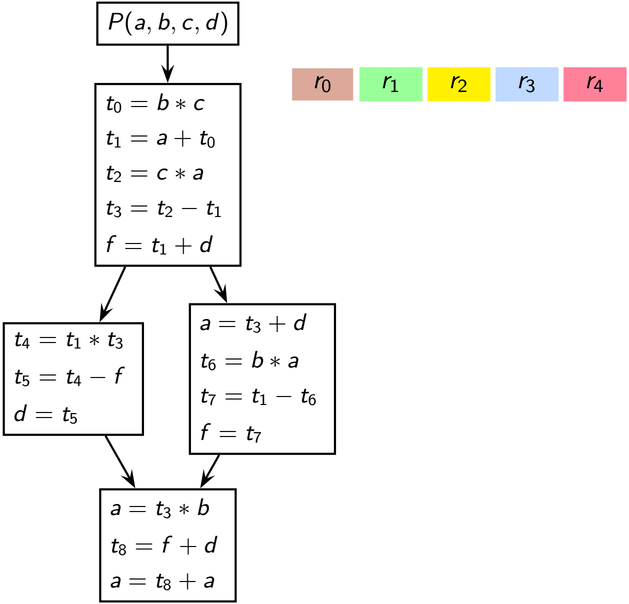
$t_0 = b * c$
$t_1 = a + t_0$
$t_2 = c * a$
$t_3 = t_2 - t_1$
$f = t_1 + d$

$t_4 = t_1 * t_3$
$t_5 = t_4 - f$
$d = t_5$

$a = t_3 + d$
$t_6 = b * a$
$t_7 = t_1 - t_6$
$f = t_7$

$a = t_3 * b$
$t_8 = f + d$
$a = t_8 + a$

Program After Global Register Allocation



$P(a, b, c, d)$

$t_0 = b * c$
$t_1 = a + t_0$
$t_2 = c * a$
$t_3 = t_2 - t_1$
$f = t_1 + d$

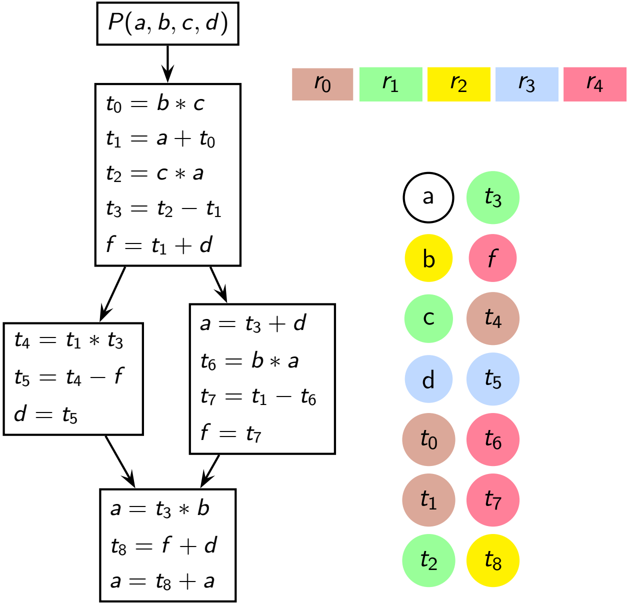
$t_4 = t_1 * t_3$
$t_5 = t_4 - f$
$d = t_5$

$a = t_3 + d$
$t_6 = b * a$
$t_7 = t_1 - t_6$
$f = t_7$

$a = t_3 * b$
$t_8 = f + d$
$a = t_8 + a$

r_0 r_1 r_2 r_3 r_4

Program After Global Register Allocation



$P(a, b, c, d)$

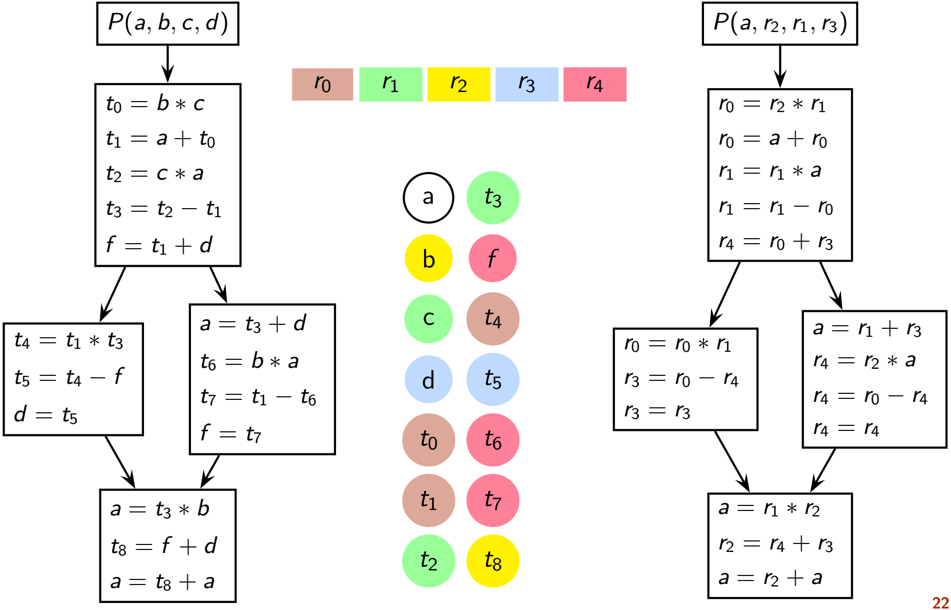
$t_0 = b * c$
$t_1 = a + t_0$
$t_2 = c * a$
$t_3 = t_2 - t_1$
$f = t_1 + d$

$t_4 = t_1 * t_3$
$t_5 = t_4 - f$
$d = t_5$

$a = t_3 + d$
$t_6 = b * a$
$t_7 = t_1 - t_6$
$f = t_7$

$a = t_3 * b$
$t_8 = f + d$
$a = t_8 + a$

Program After Global Register Allocation



$P(a, b, c, d)$

$t_0 = b * c$
$t_1 = a + t_0$
$t_2 = c * a$
$t_3 = t_2 - t_1$
$f = t_1 + d$

$t_4 = t_1 * t_3$
$t_5 = t_4 - f$
$d = t_5$

$a = t_3 + d$
$t_6 = b * a$
$t_7 = t_1 - t_6$
$f = t_7$

$a = t_3 * b$
$t_8 = f + d$
$a = t_8 + a$

r_0 r_1 r_2 r_3 r_4

$P(a, r_2, r_1, r_3)$

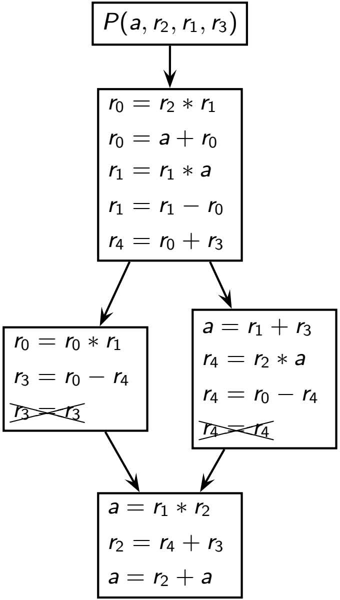
$r_0 = r_2 * r_1$
$r_0 = a + r_0$
$r_1 = r_1 * a$
$r_1 = r_1 - r_0$
$r_4 = r_0 + r_3$

$r_0 = r_0 * r_1$
$r_3 = r_0 - r_4$
$r_3 = r_3$

$a = r_1 + r_3$
$r_4 = r_2 * a$
$r_4 = r_0 - r_4$
$r_4 = r_4$

$a = r_1 * r_2$
$r_2 = r_4 + r_3$
$a = r_2 + a$

Program After Peephole Optimization



$P(a, r_2, r_1, r_3)$

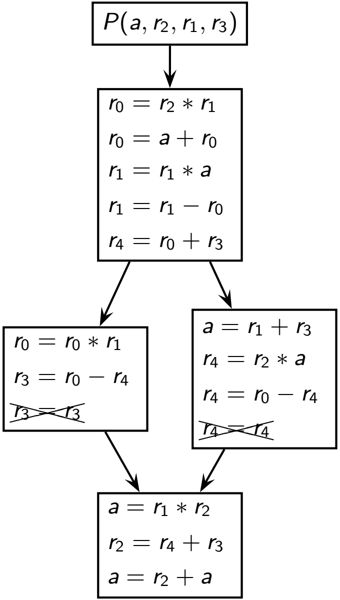
$r_0 = r_2 * r_1$
$r_0 = a + r_0$
$r_1 = r_1 * a$
$r_1 = r_1 - r_0$
$r_4 = r_0 + r_3$

$r_0 = r_0 * r_1$
$r_3 = r_0 - r_4$
~~$r_3 = r_3$~~

$a = r_1 + r_3$
$r_4 = r_2 * a$
$r_4 = r_0 - r_4$
~~$r_4 = r_4$~~

$a = r_1 * r_2$
$r_2 = r_4 + r_3$
$a = r_2 + a$

Program After Peephole Optimization



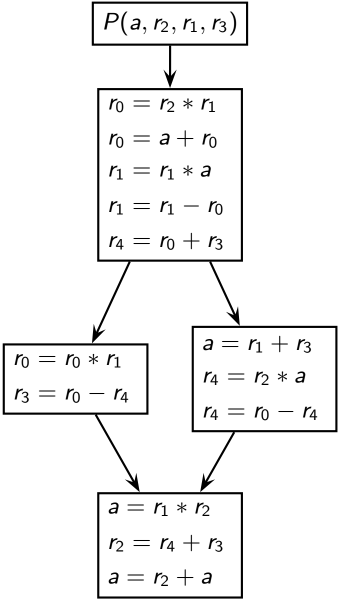
$P(a, r_2, r_1, r_3)$

$r_0 = r_2 * r_1$
$r_0 = a + r_0$
$r_1 = r_1 * a$
$r_1 = r_1 - r_0$
$r_4 = r_0 + r_3$

$r_0 = r_0 * r_1$
$r_3 = r_0 - r_4$
$\xcancel{r_3 = r_3}$

$a = r_1 + r_3$
$r_4 = r_2 * a$
$r_4 = r_0 - r_4$
$\xcancel{r_4 = r_4}$

$a = r_1 * r_2$
$r_2 = r_4 + r_3$
$a = r_2 + a$



$P(a, r_2, r_1, r_3)$

$r_0 = r_2 * r_1$
$r_0 = a + r_0$
$r_1 = r_1 * a$
$r_1 = r_1 - r_0$
$r_4 = r_0 + r_3$

$r_0 = r_0 * r_1$
$r_3 = r_0 - r_4$

$a = r_1 + r_3$
$r_4 = r_2 * a$
$r_4 = r_0 - r_4$

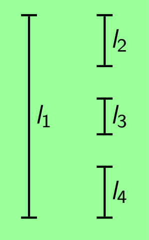
$a = r_1 * r_2$
$r_2 = r_4 + r_3$
$a = r_2 + a$

Live Range Spilling

- Spilling a live range l involves keeping the variable of l in the memory,
    - For RISC architectures: load in a register for every read, store back in the memory for every write
    - For CISC architectures: access directly from memory

Live Range Spilling

- Spilling a live range l involves keeping the variable of l in the memory,
    - For RISC architectures: load in a register for every read, store back in the memory for every write
    - For CISC architectures: access directly from memory
- Spilling is necessary if the number of interfering live ranges at a program point exceeds the number of registers



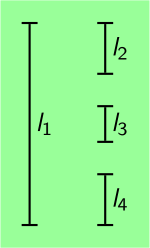
- The degree of l_1 is 3 but at no point does it exceed 2
- 2 registers are sufficient without needing spilling

Live Range Splitting

- Splitting a live range l involves creating smaller live ranges $l_1, \ldots, l_k$ such that $D(l_i) \leq D(l)$, $1 \leq i \leq k$
    - Live ranges l_i participate in graph colouring and may get a colour if $D(l_i) < D(l)$

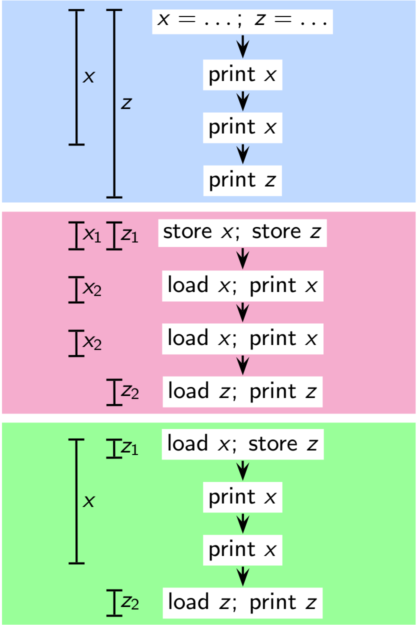
Live Range Splitting

- Splitting a live range l involves creating smaller live ranges $l_1, \ldots, l_k$ such that $D(l_i) \leq D(l)$, $1 \leq i \leq k$
    - Live ranges l_i participate in graph colouring and may get a colour if $D(l_i) < D(l)$
- Splitting l is useful when it *contains* some live range l' completely and at some point $D(l')$ is smaller than the number of registers



Splitting cannot help us colour this program with a single colour

Spilling Vs Splitting



$x = \ldots;\ z = \ldots$

print x

print x

print z

Original program

store x; store z

load x; print x

load x; print x

load z; print z

Splitting x and z leads
to 3 loads and 2 stores

load x; store z

print x

print x

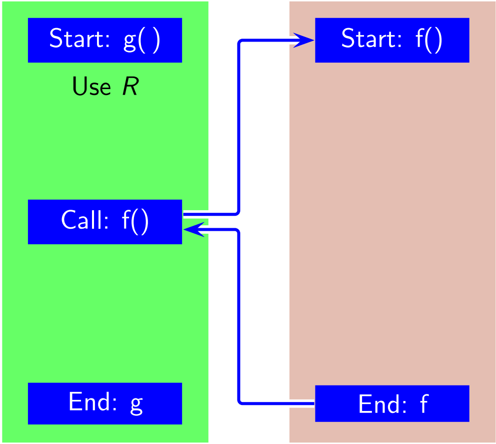
load z; print z

Splitting only z leads
to 2 loads and 1 store

No difference between
spitting or spilling z

Managing Registers Across Calls

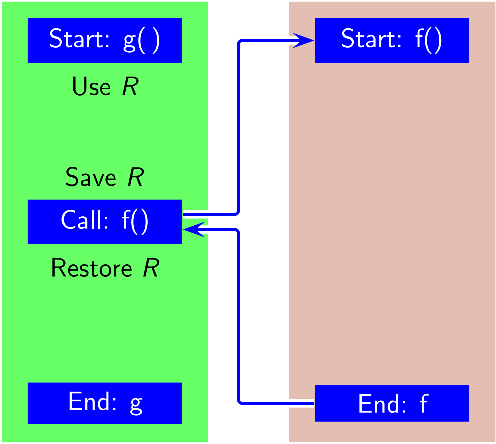
Managing Registers Across Calls



Two options to manage R across the call

- Procedure g saves it before the call and restores it after the call
- Procedure f saves it at the start and restores it the end

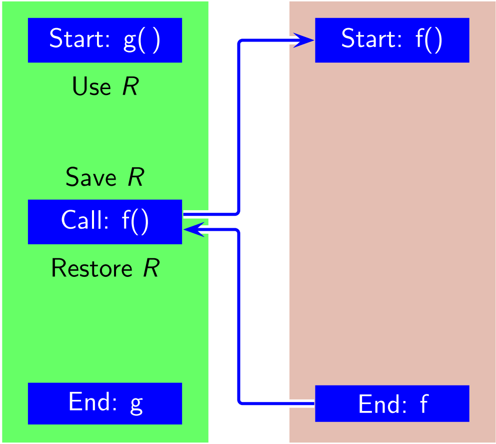
Managing Registers Across Calls



If procedure g saves R before the call and restores it after the call

- It does not know if procedure f really needs R
- However, it knows if the value in R is needed across the call

Managing Registers Across Calls



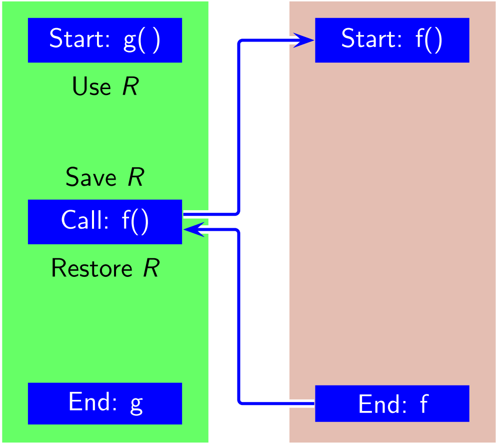
If procedure g saves R before the call and restores it after the call

- It does not know if procedure f really needs R
- However, it knows if the value in R is needed across the call

Save and restore would be wasteful if

- f does not need R, or
- the value in R is not needed across the call

Managing Registers Across Calls



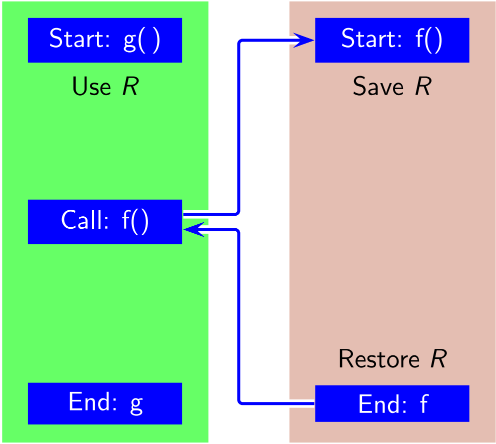
If procedure g saves R before the call and restores it after the call

- It does not know if procedure f really needs R
- However, it knows if the value in R is needed across the call

Save and restore would be wasteful if

- f does not need R, or
  Unavoidable
- the value in R is not needed across the call
  Avoidable

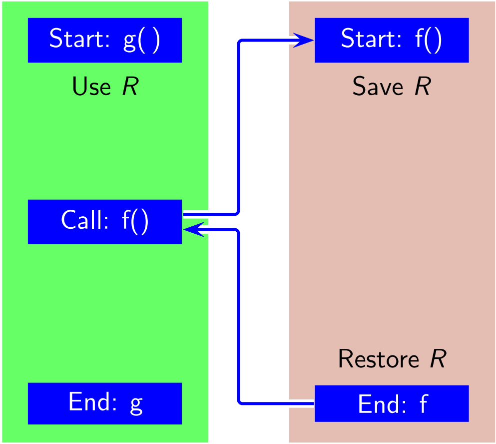
Managing Registers Across Calls



If procedure f saves R at the start and restores it at the end

- It does not know if procedure g contains a value needed across the call
- However, it knows if R is needed within f

Managing Registers Across Calls



Start: g( )

Use R

Call: f()

End: g

Start: f()

Save R

Restore R

End: f

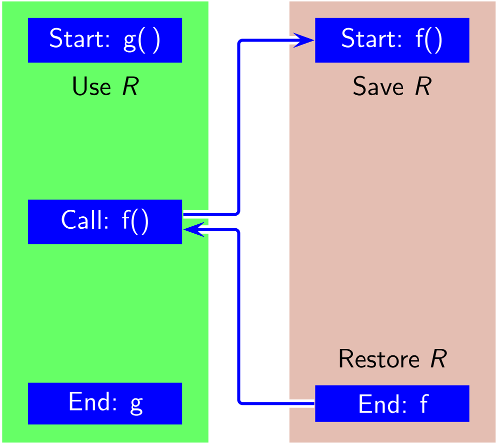
If procedure f saves R at the start and restores it at the end

- It does not know if procedure g contains a value needed across the call
- However, it knows if R is needed within f

Save and restore would be wasteful if

- the value in R is not needed across the call, or
- f does not need R

Managing Registers Across Calls



If procedure f saves R at the start and restores it at the end

- It does not know if procedure g contains a value needed across the call
- However, it knows if R is needed within f

Save and restore would be wasteful if

- the value in R is not needed across the call, or **Unavoidable**
- f does not need R **Avoidable**

Caller-Saved and Callee-Saved Registers

- Caller-saved register.
    - Saving is at the discretion of the caller
    - Callee can use it without the fear of overwriting useful data
    - Also known as call-clobbered register
- Callee-saved register.
    - Saving is at the discretion of the callee
    - Caller can use it without the fear of overwriting useful data
    - Also known as call-preserved register

Caller-Saved and Callee-Saved Registers

- Caller-saved register.
    - Saving is at the discretion of the caller
    - Callee can use it without the fear of overwriting useful data
    - Also known as call-clobbered register
- Callee-saved register.
    - Saving is at the discretion of the callee
    - Caller can use it without the fear of overwriting useful data
    - Also known as call-preserved register
- How to use these registers in a procedure?

Caller-Saved and Callee-Saved Registers

- Caller-saved register.
  - Saving is at the discretion of the caller
  - Callee can use it without the fear of overwriting useful data
  - Also known as call-clobbered register
- Callee-saved register.
  - Saving is at the discretion of the callee
  - Caller can use it without the fear of overwriting useful data
  - Also known as call-preserved register
- How to use these registers in a procedure?
  - Use a caller-saved register R for values that are not live across a call
    R is not saved by the callee (convention)
    nor by the caller because the value is not needed

Caller-Saved and Callee-Saved Registers

- Caller-saved register.
  - Saving is at the discretion of the caller
  - Callee can use it without the fear of overwriting useful data
  - Also known as call-clobbered register
- Callee-saved register.
  - Saving is at the discretion of the callee
  - Caller can use it without the fear of overwriting useful data
  - Also known as call-preserved register
- How to use these registers in a procedure?
  - Use a caller-saved register R for values that are not live across a call
    R is not saved by the callee (convention)
    nor by the caller because the value is not needed
  - Use a callee-saved register R for values that are live across a call
    R is not saved by the caller (convention)
    saved by the callee only if the value is needed

IIT Bombay
cs302: Implementation of Programming Languages

Topic:

Code Generation

Section:

Global Register Allocation

Managing Registers Across Calls

Registers Usage in sclp

Instruction Selection

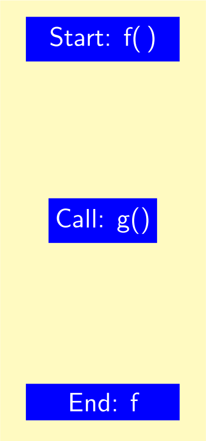
Caller-Saved and Callee-Saved Registers

- Caller-saved register.
  - Saving is at the discretion of the caller
  - Callee can use it without the fear of overwriting useful data
  - Also known as call-clobbered register
- Callee
  - S
  - C
  - A
- How t
  - U oss a call
    R
    nor by the caller because the value is not needed
  - Use a callee-saved register R for values that are live across a call
    R is not saved by the caller (convention)
    saved by the callee only if the value is needed

This convention is decided by the architecture (and not by a compiler) to facilitate separate compilation

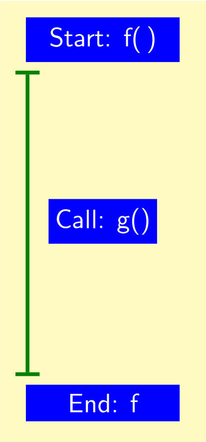
Different object modules, in particular libraries, may be compiled by different compilers and must follow a uniform convention

Integrating Caller-Saved and Callee-Saved Registers Within Graph Colouring



- Let callee-saved and caller-saved registers be denoted by synthetic live ranges coloured green and red respectively

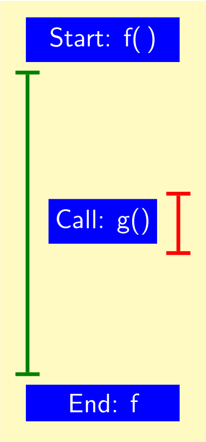
Integrating Caller-Saved and Callee-Saved Registers Within Graph Colouring



- Let callee-saved and caller-saved registers be denoted by synthetic live ranges coloured green and red respectively
- In f, a callee-saved registered is assumed to be occupying a value that is live across a call of f in its callers

  Construct a green live range spanning the entire body of f (indicating that it is not available in the body of f because it is used in the callers of f)

Integrating Caller-Saved and Callee-Saved Registers Within Graph Colouring

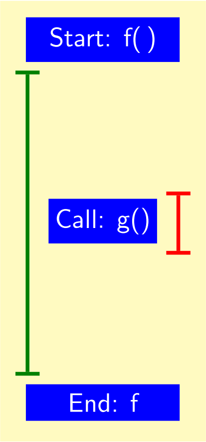


- Let callee-saved and caller-saved registers be denoted by synthetic live ranges coloured green and red respectively
- In f, a callee-saved registered is assumed to be occupying a value that is live across a call of f in its callers

  Construct a green live range spanning the entire body of f (indicating that it is not available in the body of f because it is used in the callers of f)
- In f, a caller-saved registered is assumed to be occupying a value that is not live across calls within f

  Construct a red live range spanning the calls in f (indicating that it is not available across the call because it is used in the callees of f)

Integrating Caller-Saved and Callee-Saved Registers Within Graph Colouring

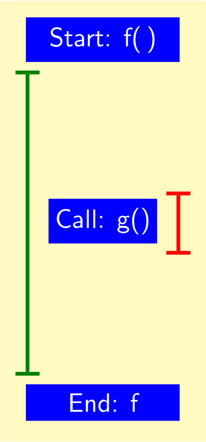


- Let callee-saved and caller-saved registers be denoted by synthetic live ranges coloured green and red respectively
- In f, a callee-saved registered is assumed to be occupying a value that is live across a call of f in its callers

  Construct a green live range spanning the entire body of f (indicating that it is not available in the body of f because it is used in the callers of f)
- In f, a caller-saved registered is assumed to be occupying a value that is not live across calls within f

  Construct a red live range spanning the calls in f (indicating that it is not available across the call because it is used in the callees of f)
- Construct the interference graph with these additional live ranges

Integrating Caller-Saved and Callee-Saved Registers Within Graph Colouring

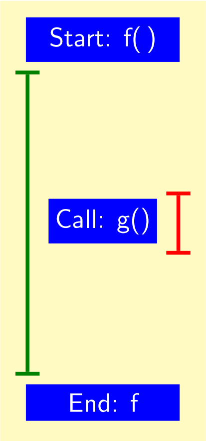


- Let callee-saved and caller-saved registers be denoted by synthetic live ranges coloured green and red respectively
- In f, a callee-saved registered is assumed to be occupying a value that is live across a call of f in its callers

  Construct a green live range spanning the entire body of f (indicating that it is not available in the body of f because it is used in the callers of f)
- In f, a caller-saved registered is assumed to be occupying a value that is not live across calls within f

  Construct a red live range spanning the calls in f (indicating that it is not available across the call because it is used in the callees of f)
- Construct the interference graph with these additional live ranges
- Colour the graph with the constraint that the red live ranges cannot be spilled/split.

Integrating Caller-Saved and Callee-Saved Registers Within Graph Colouring



- Let callee-saved and caller-saved registers be denoted by synthetic li
- I alue t
  C
  (is
  u
- I lue t
  C hat
  it es
  o
- C ges
- Colour the graph with the constraint that the red live ranges cannot be spilled/split.

We may spill/split green or other live ranges (except red) if we cannot colour the graph

The reason red live range cannot be spilled or split is that it is used in the callee procedure *g* and *f* does not have access to the value; in *f*, we can save a value before the call and restore it but it does not amount to spilling the red live range in the callee *g*

The green live range can be spilled by saving the value at the start of *f* and restoring at the end of *f* (hence spill cost 2)

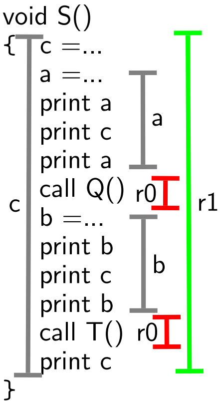
If no live range needs to spill/split, we have avoided all saves and restores across the calls to *f* and within *f*

Example of Register Management Across Calls (2 Registers)

```
void S()
{   c =...
    a =...
    print a
    print c
    print a
    call Q()
    b =...
    print b
    print c
    print b
    call T()
    print c
}
```

```
void Q()
{   f =...
    d =...
    print d
    print f
    print d
    e =...
    print e
    print f
    print e
    print f
}
```

Example of Register Management Across Calls (2 Registers)



```
void S()
{   c =...
    a =...
    print a
    print c
    print a
    call Q()
    b =...
    print b
    print c
    print b
    call T()
    print c
}
```

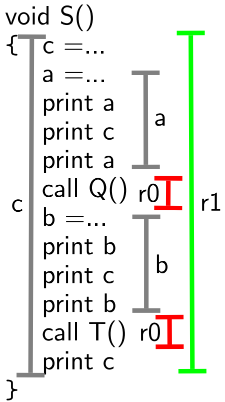
```
void Q()
{   f =...
    d =...
    print d
    print f
    print d
    e =...
    print e
    print f
    print e
    print f
}
```

Identify live ranges in procedure *S*

Add a green live range to represent that a caller of *S* may be freely using the callee-saved register r1 (which should be saved by *S* in its role as a callee)

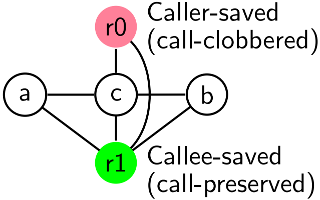
Add a red live range across each call to represent that a callee of *S* may be freely using the caller-saved register r0 (which should be saved by *S* in its role as a caller)

Example of Register Management Across Calls (2 Registers)



```
void S()
{   c =...
    a =...
    print a
    print c
    print a
    call Q()
    b =...
    print b
    print c
    print b
    call T()
    print c
}
```

```
void Q()
{   f =...
    d =...
    print d
    print f
    print d
    e =...
    print e
    print f
    print e
    print f
}
```



Caller-saved (call-clobbered)

Callee-saved (call-preserved)

Identify live ranges in procedure *S*

Add a green live range to represent that a caller of *S* may be freely using the callee-saved register r1 (which should be saved by *S* in its role as a callee)

Add a red live range across each call to represent that a callee of *S* may be freely using the caller-saved register r0 (which should be saved by *S* in its role as a caller)

Construct the interference graph for *S*

Example of Register Management Across Calls (2 Registers)

```
void S()
{   c =...
    a =...
    print a
    print c
    print a
    call Q()
    b =...
    print b
    print c
    print b
    call T()
    print c
}
```

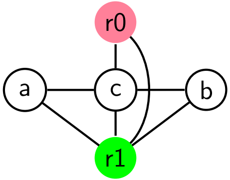
```
void Q()
{   f =...
    d =...
    print d
    print f
    print d
    e =...
    print e
    print f
    print e
    print f
}
```

Identify live ranges in procedure *S*

Add a green live range to represent that a caller of *S* may be freely using the callee-saved register r1 (which should be saved by *S* in its role as a callee)

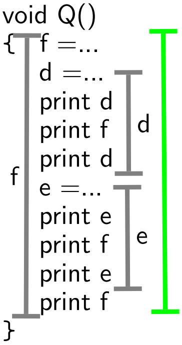
Add a red live range across each call to represent that a callee of *S* may be freely using the caller-saved register r0 (which should be saved by *S* in its role as a caller)

Construct the interference graph for *S*



Example of Register Management Across Calls (2 Registers)

```
void S()
{   c =...
    a =...
    print a
    print c
    print a
    call Q()
    b =...
    print b
    print c
    print b
    call T()
    print c
}
```

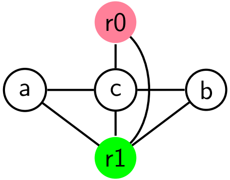


```
void Q()
{   f =...
    d =...
    print d
    print f
    print d
    e =...
    print e
    print f
    print e
    print f
}
```

Identify live ranges in procedure *Q*

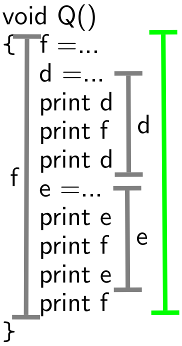
Add a green live range to represent that a caller of *Q* may be freely using the callee-saved register r1 (which should be saved by *Q* in its role as a callee)

No call in *Q* so no red live range (or it does not interfere with any live range)



Example of Register Management Across Calls (2 Registers)

```
void S()
{   c =...
    a =...
    print a
    print c
    print a
    call Q()
    b =...
    print b
    print c
    print b
    call T()
    print c
}
```



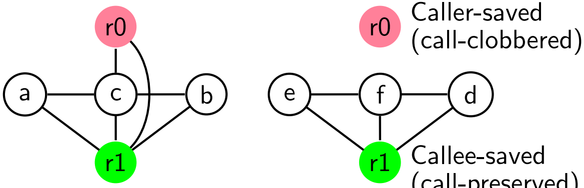
```
void Q()
{   f =...
    d =...
    print d
    print f
    print d
    e =...
    print e
    print f
    print e
    print f
}
```

Identify live ranges in procedure Q

Add a green live range to represent that a caller of Q may be freely using the callee-saved register r1 (which should be saved by Q in its role as a callee)

No call in Q so no red live range (or it does not interfere with any live range)

Construct the interference graph for Q



Example of Register Management Across Calls (2 Registers)

```
void S()
{   c =...
    a =...
    print a
    print c
    print a
    call Q()
    b =...
    print b
    print c
    print b
    call T()
    print c
}
```

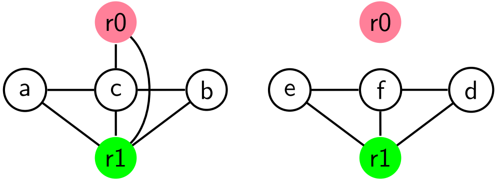
```
void Q()
{   f =...
    d =...
    print d
    print f
    print d
    e =...
    print e
    print f
    print e
    print f
}
```

Identify live ranges in procedure *Q*

Add a green live range to represent that a caller of *Q* may be freely using the callee-saved register r1 (which should be saved by *Q* in its role as a callee)

No call in *Q* so no red live range (or it does not interfere with any live range)

Construct the interference graph for *Q*

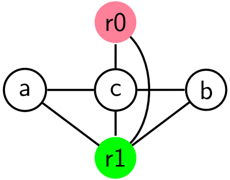


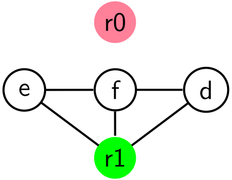
Example of Register Management Across Calls (2 Registers)

```
void S()
{   c =...
    a =...
    print a
    print c
    print a
    call Q()
    b =...
    print b
    print c
    print b
    call T()
    print c
}
```

```
void Q()
{   f =...
    d =...
    print d
    print f
    print d
    e =...
    print e
    print f
    print e
    print f
}
```

Cannot colour the interference graph of *S* with two colours so we need to spill some live ranges





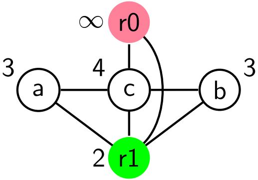
Example of Register Management Across Calls (2 Registers)

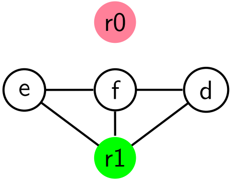
```
void S()
{   c =...
    a =...
    print a
    print c
    print a
    call Q()
    b =...
    print b
    print c
    print b
    call T()
    print c
}
```

```
void Q()
{   f =...
    d =...
    print d
    print f
    print d
    e =...
    print e
    print f
    print e
    print f
}
```

Cannot colour the interference graph of *S* with two colours so we need to spill some live ranges

Compute the spill costs





Example of Register Management Across Calls (2 Registers)

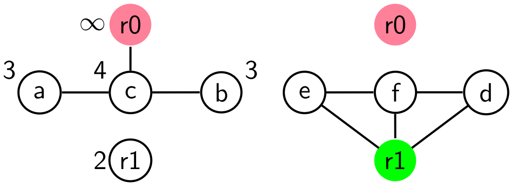
```
void S()
{   c =...
    a =...
    print a
    print c
    print a
    call Q()
    b =...
    print b
    print c
    print b
    call T()
    print c
}
```

```
void Q()
{   f =...
    d =...
    print d
    print f
    print d
    e =...
    print e
    print f
    print e
    print f
}
```

Cannot colour the interference graph of *S* with two colours so we need to spill some live ranges

Compute the spill costs

Spill r1 because it has the least spill cost



Example of Register Management Across Calls (2 Registers)

```
void S()
{   c =...
    a =...
    print a
    print c
    print a
    call Q()
    b =...
    print b
    print c
    print b
    call T()
    print c
}
```

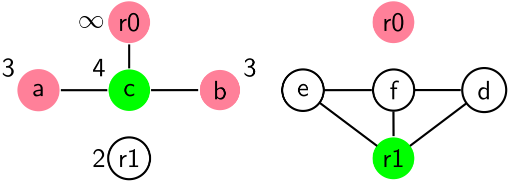
```
void Q()
{   f =...
    d =...
    print d
    print f
    print d
    e =...
    print e
    print f
    print e
    print f
}
```

Cannot colour the interference graph of *S* with two colours so we need to spill some live ranges

Compute the spill costs

Spill r1 because it has the least spill cost

Give green color to *c* and red to *a* and *b*

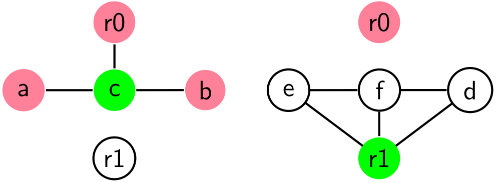


Example of Register Management Across Calls (2 Registers)

```
void S()
{   c =...
    a =...
    print a
    print c
    print a
    call Q()
    b =...
    print b
    print c
    print b
    call T()
    print c
}
```

```
void Q()
{   f =...
    d =...
    print d
    print f
    print d
    e =...
    print e
    print f
    print e
    print f
}
```

Cannot colour the interference graph of *Q* with two colours so we need to spill some live ranges



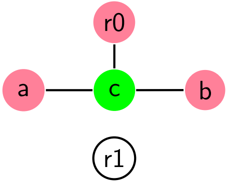
Example of Register Management Across Calls (2 Registers)

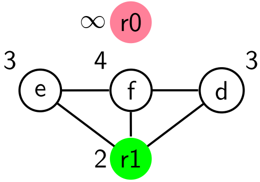
```
void S()
{   c =...
    a =...
    print a
    print c
    print a
    call Q()
    b =...
    print b
    print c
    print b
    call T()
    print c
}
```

```
void Q()
{   f =...
    d =...
    print d
    print f
    print d
    e =...
    print e
    print f
    print e
    print f
}
```

Cannot colour the interference graph of *Q* with two colours so we need to spill some live ranges

Compute the spill costs





Example of Register Management Across Calls (2 Registers)

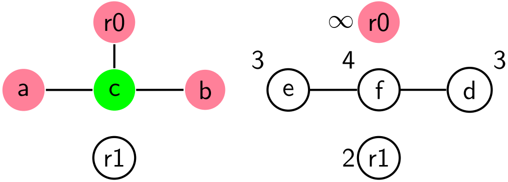
```
void S()
{   c =...
    a =...
    print a
    print c
    print a
    call Q()
    b =...
    print b
    print c
    print b
    call T()
    print c
}
```

```
void Q()
{   f =...
    d =...
    print d
    print f
    print d
    e =...
    print e
    print f
    print e
    print f
}
```

Cannot colour the interference graph of *Q* with two colours so we need to spill some live ranges

Compute the spill costs

Spill r1 because it has the least spill cost



Example of Register Management Across Calls (2 Registers)

```
void S()
{   c =...
    a =...
    print a
    print c
    print a
    call Q()
    b =...
    print b
    print c
    print b
    call T()
    print c
}
```

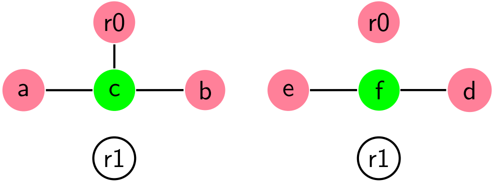
```
void Q()
{   f =...
    d =...
    print d
    print f
    print d
    e =...
    print e
    print f
    print e
    print f
}
```

Cannot colour the interference graph of *Q* with two colours so we need to spill some live ranges

Compute the spill costs

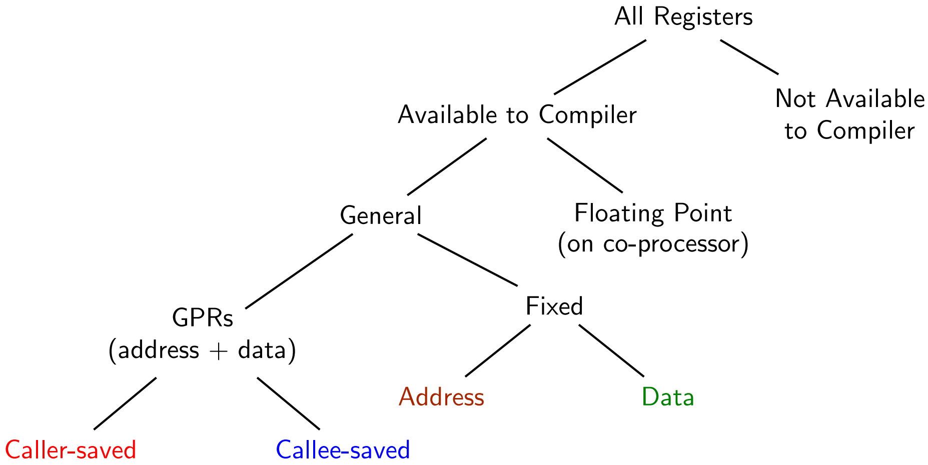
Spill r1 because it has the least spill cost

Give one color to *f* and the other color to *e* and *d*



Registers Usage in sclp

Register Categories for Spim



All Registers

Available to Compiler

Not Available to Compiler

General

Floating Point (on co-processor)

GPRs (address + data)

Fixed

Address

Data

Caller-saved

Callee-saved

Registers in Spim

Name	No.	Sizes	Use	Category
`$zero`	00	32		constant data
`at`	01	32		NA
`v0`	02	32,64	expr result	caller-saved
`v1`	03	32	function result	caller-saved
`a0`	04	32,64	argument	caller-saved
`a1`	05	32	argument	caller-saved
`a2`	06	32,64	argument	caller-saved
`a3`	07	32	argument	caller-saved
`t0`	08	32,64	temporary	caller-saved
`t1`	09	32	temporary	caller-saved
`t2`	10	32,64	temporary	caller-saved
`t3`	11	32	temporary	caller-saved
`t4`	12	32,64	temporary	caller-saved
`t5`	13	32	temporary	caller-saved
`t6`	14	32,64	temporary	caller-saved
`t7`	15	32	temporary	caller-saved

Name	No.	Sizes	Use	Category
`s0`	16	32,64	temporary	callee-saved
`s1`	17	32	temporary	callee-saved
`s2`	18	32,64	result	callee-saved
`s3`	19	32	result	callee-saved
`s4`	20	32,64	temporary	callee-saved
`s5`	21	32	temporary	callee-saved
`s6`	22	32,64	temporary	callee-saved
`s7`	23	32	temporary	callee-saved
`t8`	24	32,64	temporary	caller-saved
`t9`	25	32	temporary	caller-saved
`k0`	26	32,64		NA
`k1`	27	32		NA
`gp`	28	32,64	global pointer	address
`sp`	29	32	stack pointer	address
`fp`	30	32,64	frame pointer	address
`ra`	31	32	return address	address

Co-Processor Registers in Spim

Name	Number	Sizes
`f0`	00	32,64
`f1`	01	32
`f2`	02	32,64
`f3`	03	32
`f4`	04	32,64
`f5`	05	32
`f6`	06	32,64
`f7`	07	32
`f8`	08	32,64
`f9`	09	32
`f10`	10	32,64
`f11`	11	32
`f12`	12	32,64
`f13`	13	32
`f14`	14	32,64
`f15`	15	32

Name	Number	Sizes
`f16`	16	32,64
`f17`	17	32
`f19`	18	32,64
`f19`	19	32
`f20`	20	32,64
`f21`	21	32
`f22`	22	32,64
`f23`	23	32
`f24`	24	32,64
`f25`	25	32
`f26`	26	32,64
`f27`	27	32
`f28`	28	32,64
`f29`	29	32
`f30`	30	32,64
`f31`	31	32

Registers Used By SCLP for Storing Intermediate Results

Register		Type
Name	Pair	
v0		int
v1		int
t0		int
t1		int
t2		int
t3		int
t4		int
t5		int
t6		int
t7		int
t8		int
t9		int

Register		Type
Name	Pair	
s0		int
s1		int
s2		int
s3		int
s4		int
s5		int
s6		int
s7		int
f0	f0,f1	float
f2	f2,f3	float
f4	f4,f5	float
f6	f5,f6	float

Register		Type
Name	Pair	
f8	f8,f9	float
f10	f10,f11	float
f12	f12,f13	float
f14	f14,f15	float
f16	f16,f17	float
f18	f18,f19	float
f20	f20,f21	float
f22	f22,f23	float
f24	f24,f25	float
f26	f26,f27	float
f28	f28,f29	float
f30	f30,f31	float

Registers Used By SCLP for Storing Intermediate Results

Designated for function result, hence ignored

Register		Type
Name	Pair	
v0		int
v1		int
t0		int
t1		int
t2		int
t3		int
t4		int
t5		int
t6		int
t7		int
t8		int
t9		int

Register		Type
Name	Pair	
s0		int
s1		int
s2		int
s3		int
s4		int
s5		int
s6		int
s7		int
f0	f0,f1	float
f2	f2,f3	float
f4	f4,f5	float
f6	f5,f6	float

Register		Type
Name	Pair	
f8	f8,f9	float
f10	f10,f11	float
f12	f12,f13	float
f14	f14,f15	float
f16	f16,f17	float
f18	f18,f19	float
f20	f20,f21	float
f22	f22,f23	float
f24	f24,f25	float
f26	f26,f27	float
f28	f28,f29	float
f30	f30,f31	float

Registers Used By SCLP for Storing Intermediate Results

Designated for function result, hence ignored

Register		Type
Name	Pair	
v0		int
t0		int
t1		int
t2		int
t3		int
t4		int
t5		int
t6		int
t7		int
t8		int
t9		int

Register		Type
Name	Pair	
s0		int
s1		int
s2		int
s3		int
s4		int
s5		int
s6		int
s7		int
f2	f2,f3	float
f4	f4,f5	float
f6	f5,f6	float

Register		Type
Name	Pair	
f8	f8,f9	float
f10	f10,f11	float
f12	f12,f13	float
f14	f14,f15	float
f16	f16,f17	float
f18	f18,f19	float
f20	f20,f21	float
f22	f22,f23	float
f24	f24,f25	float
f26	f26,f27	float
f28	f28,f29	float
f30	f30,f31	float

Register Allocation Policy Used by sclp

Very simple strategy

- No register is occupied across any assignment in the source program

  The result is stored in memory for the LHS variable and the result register is freed
- Within an expression, values of source variables are loaded into registers
- Intermediate values within an expression (stored in a temporary variable) are assigned a register

  The result of a ternary expression is a “saved” temporary which is treated like a source variable
- Getting a new register
  - When a temporary is assigned a register, mark it as occupied
  - When a temporary is used in a TAC statement, mark the register as free
  - To get a new register,
    - Traverse the list of registers and assign the first free register
    - Need to match the type
- Registers are chosen for a TAC statement in this order: first operand, result, second operand (because second operand may not exist)

IIT Bombay
cs302: Implementation of Programming Languages

Topic:

Code Generation

Section:

Global Register Allocation

Managing Registers Across Calls

Registers Usage in sclp

Instruction Selection

Instruction Selection

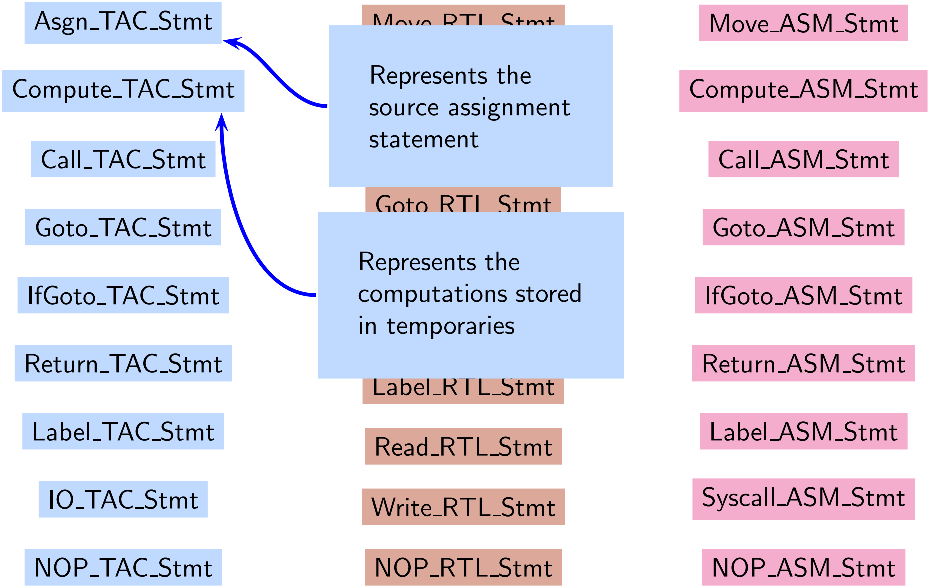
Instruction Selection in sclp

- Asgn_TAC_Stmt
- Compute_TAC_Stmt
- Call_TAC_Stmt
- Goto_TAC_Stmt
- IfGoto_TAC_Stmt
- Return_TAC_Stmt
- Label_TAC_Stmt
- IO_TAC_Stmt
- NOP_TAC_Stmt

- Move_RTL_Stmt
- Compute_RTL_Stmt
- Call_RTL_Stmt
- Goto_RTL_Stmt
- IfGoto_RTL_Stmt
- Return_RTL_Stmt
- Label_RTL_Stmt
- Read_RTL_Stmt
- Write_RTL_Stmt
- NOP_RTL_Stmt

- Move_ASM_Stmt
- Compute_ASM_Stmt
- Call_ASM_Stmt
- Goto_ASM_Stmt
- IfGoto_ASM_Stmt
- Return_ASM_Stmt
- Label_ASM_Stmt
- Syscall_ASM_Stmt
- NOP_ASM_Stmt

Instruction Selection in sclp



Asgn_TAC_Stmt
Compute_TAC_Stmt
Call_TAC_Stmt
Goto_TAC_Stmt
IfGoto_TAC_Stmt
Return_TAC_Stmt
Label_TAC_Stmt
IO_TAC_Stmt
NOP_TAC_Stmt

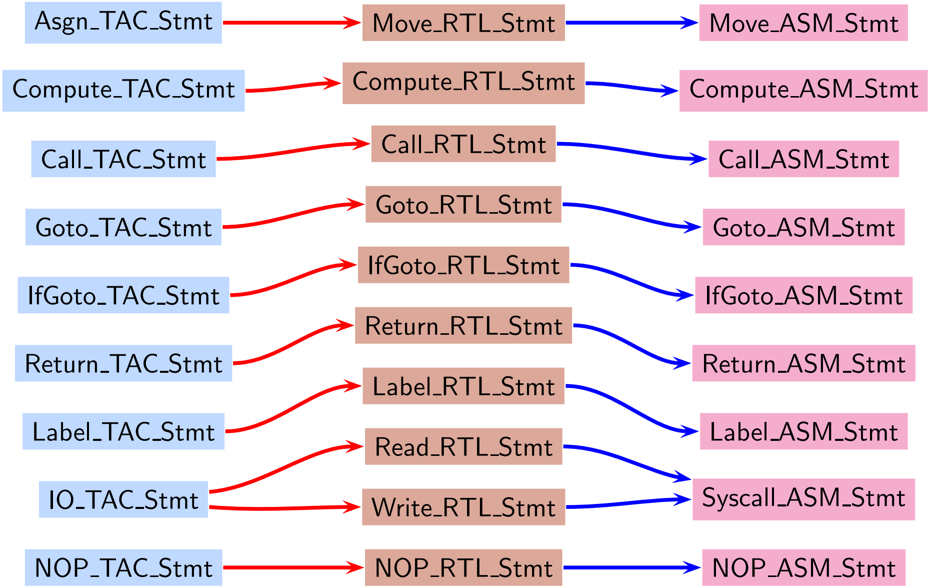
Move_RTL_Stmt
Goto_RTL_Stmt
Label_RTL_Stmt
Read_RTL_Stmt
Write_RTL_Stmt
NOP_RTL_Stmt

Move_ASM_Stmt
Compute_ASM_Stmt
Call_ASM_Stmt
Goto_ASM_Stmt
IfGoto_ASM_Stmt
Return_ASM_Stmt
Label_ASM_Stmt
Syscall_ASM_Stmt
NOP_ASM_Stmt

Represents the source assignment statement

Represents the computations stored in temporaries

Instruction Selection in sclp



Integrated Instruction Selection and Register Allocation Algorithms

For expression trees

- Sethi-Ullman Algorithm
    - Simple machine model (handles RISC architectures well)
    - Optimal in terms of the number of instructions with the minimum number of registers and minimum number of stores
    - Linear in the size of the expression tree
- Aho-Johnson Algorithm
    - Very general machine model (handles CISC architectures also well)
    - Optimal in terms of the cost of execution
    - Linear in the size of the expression tree
      (exponential in the arity of instructions which is bounded by a small constant, say 3 or 4)

Integrated Instruction Selection and Register Allocation Algorithms

For expression trees

- Sethi-Ullman Algorithm
  - Simple machine model (handles RISC architectures well)
  - Optimal in terms of the number of instructions with the minimum number of registers and minimum number of stores
  - Linear in the size of the expression tree
- Aho-Johnson Algorithm
  - Very general machine model (handles CISC architectures also well)
  - Optimal in terms of the cost of execution
  - Linear in the size of the expression tree
    (exponential in the arity of instructions which is bounded by a small constant, say 3 or 4)

For other IR statements, instruction selection is relatively easier and simple methods work well

Integrated Instruction Selection and Register Allocation Algorithms

For expression trees

- Sethi-Ullman Algorithm

  We will cover this

  - Simple machine model (handles RISC architectures well)
  - Optimal in terms of the number of instructions with the minimum number of registers and minimum number of stores
  - Linear in the size of the expression tree
- Aho-Johnson Algorithm

  No time for this

  - Very general machine model (handles CISC architectures also well)
  - Optimal in terms of the cost of execution
  - Linear in the size of the expression tree
    (exponential in the arity of instructions which is bounded by a small constant, say 3 or 4)

For other IR statements, instruction selection is relatively easier and simple methods work well

Sethi-Ullman Algorithm: Target Model

- A finite set of registers $r_0, r_1, \ldots, r_k$
- Countable memory locations
- Simple machine instructions
  - Load instruction $r \leftarrow m$
  - Store instruction $m \leftarrow r$
  - Compute instructions
    - $r \leftarrow r\ op\ m$ (result and left operands are same, right operand is in memory)
    - $r_1 \leftarrow r_1\ op\ r_2$ (result and left operands are same, right operand is in register)

Sethi-Ullman Algorithm: Input IR

- Expression tree (AST of expressions)
  - no control flow (so no ternary expression)
  - no assignments to source variables (so no side effects),
  - no function calls,
  - no sharing of value (so no DAGs, only trees)
- Algebraic properties such as commutativity and associative not assumed in the basic algorithm

  Extended algorithm handles them

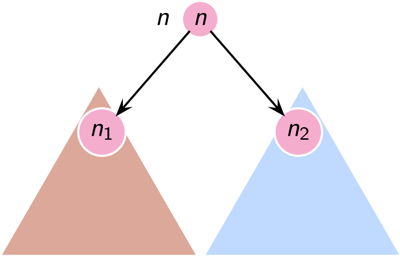
Handling a DAG

- Generating code to evaluate the shared subexpression only once could enhance efficiency
- Sethi-Ullman algorithm can be made to handle this in the following manner
    - Treat the shared subexpression as a separate expression tree, generate code for it using the Sethi-Ullman algorithm and save the result in a temporary
    - Convert the input DAG into a tree by replacing the subtree by the name of the temporary and replicate it
    - Generate the code using Sethi-Ullman algorithm

The Key Idea Behind Sethi-Ullman Algorithm

The register usage in the code fragment for a tree rooted at n can be described by

- $R(n)$: The number of registers used by the code
- $L(n)$: The number of registers live outside the code for n (i.e., the intermediate results that are required outside the code for n)
- The algorithm minimizes $R(n)$ to avoid storing intermediate results



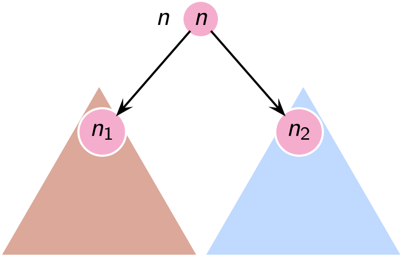
If the code computes n_1 first, then

$$R(n) = \max(R(n_1), L(n_1) + R(n_2))$$

If the code computes n_2 first, then

$$R(n) = \max(R(n_2), L(n_2) + R(n_1))$$

The Key Idea Behind Sethi-Ullman Algorithm



If the code computes n_1 first, then

$$R(n) = \max(R(n_1), L(n_1) + R(n_2))$$

If the code computes n_2 first, then

$$R(n) = \max(R(n_2), L(n_2) + R(n_1))$$

In order to minimize $R(n)$,

1. Minimize $L(n_1)$ and $L(n_2)$

   Minimizing to 0 would introduce a store so minimize to 1

2. Decide whether to evaluate n_1 first or n_2 first

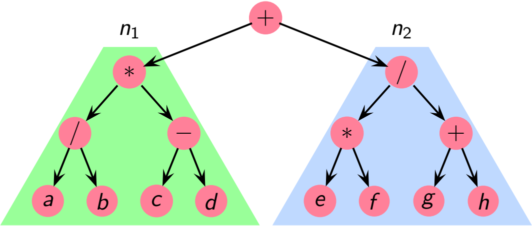
Key Idea #1: Contiguous Evaluation Minimizes $L(n)$ to 1

- Contiguous Evaluation. Evaluate n_1 completely before evaluating n_2 or vice-versa
    - If we evaluate a subtree completely, we need to hold only the final result in a register during the evaluation of the other subtrees
    - If we do not evaluate a subtree completely before moving to the other subtree, we may have to hold multiple intermediate results in a register during the evaluation of the other subtree
    - This increases the need of registers
- Strongly Contiguous Evaluation. All subtrees of n_1 and n_2 are also evaluated contiguously

Key Idea #1: Contiguous Evaluation Minimizes $L(n)$ to 1

Instructions

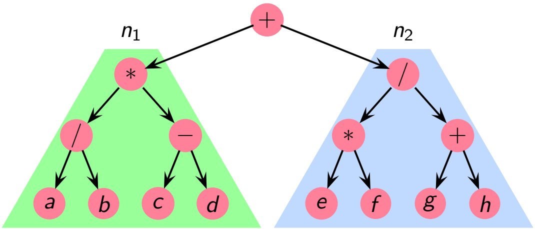
$r \leftarrow r\ op\ m$
$r_1 \leftarrow r_1\ op\ r_2$
$r \leftarrow m$
$m \leftarrow r$



Key Idea #1: Contiguous Evaluation Minimizes $L(n)$ to 1

Instructions

$r \leftarrow r \; op \; m$
$r_1 \leftarrow r_1 \; op \; r_2$
$r \leftarrow m$
$m \leftarrow r$



$r_0 \leftarrow a$
$r_0 \leftarrow r_0 / b$
$r_1 \leftarrow c$
$r_1 \leftarrow r_1 - d$
$r_0 \leftarrow r_0 * r_1$

$r_1 \leftarrow e$
$r_1 \leftarrow r_1 * f$
$r_2 \leftarrow g$
$r_2 \leftarrow r_2 + h$
$r_1 \leftarrow r_1 / r_2$

$r_0 \leftarrow r_0 + r_1$

$r_0 \leftarrow a$
$r_0 \leftarrow r_0 / b$
$r_1 \leftarrow c$
$r_1 \leftarrow r_1 - d$

$r_2 \leftarrow e$
$r_2 \leftarrow r_2 * f$
$r_3 \leftarrow g$
$r_3 \leftarrow r_3 + h$

$r_0 \leftarrow r_0 * r_1$

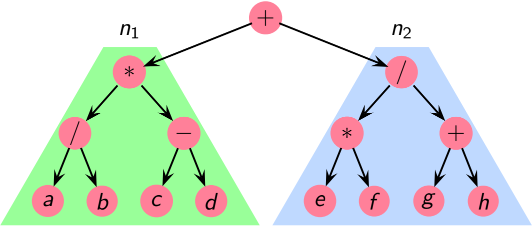
$r_2 \leftarrow r_2 / r_3$

$r_0 \leftarrow r_0 + r_2$

Key Idea #1: Contiguous Evaluation Minimizes $L(n)$ to 1

Instructions

$r \leftarrow r \; op \; m$
$r_1 \leftarrow r_1 \; op \; r_2$
$r \leftarrow m$
$m \leftarrow r$



Contiguous

$L(n_1) = 1$

$r_0 \leftarrow a$
$r_0 \leftarrow r_0 / b$
$r_1 \leftarrow c$
$r_1 \leftarrow r_1 - d$
$r_0 \leftarrow r_0 * r_1$

$r_1 \leftarrow e$
$r_1 \leftarrow r_1 * f$
$r_2 \leftarrow g$
$r_2 \leftarrow r_2 + h$
$r_1 \leftarrow r_1 / r_2$

$r_0 \leftarrow r_0 + r_1$

$r_0 \leftarrow a$
$r_0 \leftarrow r_0 / b$
$r_1 \leftarrow c$
$r_1 \leftarrow r_1 - d$

$r_2 \leftarrow e$
$r_2 \leftarrow r_2 * f$
$r_3 \leftarrow g$
$r_3 \leftarrow r_3 + h$

$r_0 \leftarrow r_0 * r_1$

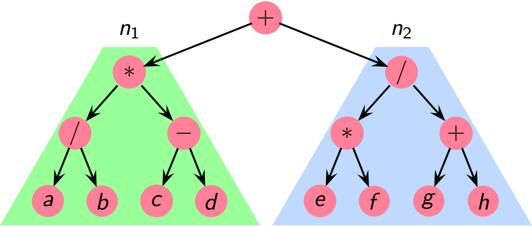
$r_2 \leftarrow r_2 / r_3$

$r_0 \leftarrow r_0 + r_2$

Key Idea #1: Contiguous Evaluation Minimizes $L(n)$ to 1

Instructions

$r \leftarrow r \; op \; m$
$r_1 \leftarrow r_1 \; op \; r_2$
$r \leftarrow m$
$m \leftarrow r$



Contiguous

$L(n_1) = 1$

$r_0 \leftarrow a$
$r_0 \leftarrow r_0 / b$
$r_1 \leftarrow c$
$r_1 \leftarrow r_1 - d$
$r_0 \leftarrow r_0 * r_1$

$r_1 \leftarrow e$
$r_1 \leftarrow r_1 * f$
$r_2 \leftarrow g$
$r_2 \leftarrow r_2 + h$
$r_1 \leftarrow r_1 / r_2$

$r_0 \leftarrow r_0 + r_1$

Non-contiguous

$L(n_1) = 2$

$r_0 \leftarrow a$
$r_0 \leftarrow r_0 / b$
$r_1 \leftarrow c$
$r_1 \leftarrow r_1 - d$

$r_2 \leftarrow e$
$r_2 \leftarrow r_2 * f$
$r_3 \leftarrow g$
$r_3 \leftarrow r_3 + h$

$r_0 \leftarrow r_0 * r_1$

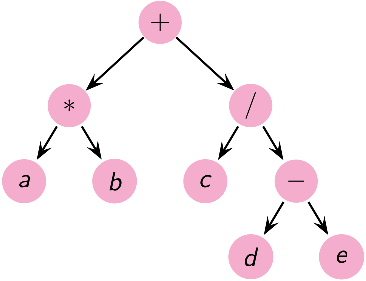
$r_2 \leftarrow r_2 / r_3$

$r_0 \leftarrow r_0 + r_2$

Key Idea #2: The Minimized $L(n_1)$ Can be "Absorbed" by Changing the Order of Evaluation

Instructions

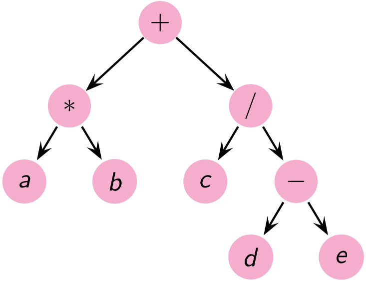
$r \leftarrow r\ op\ m$
$r_1 \leftarrow r_1\ op\ r_2$
$r \leftarrow m$
$m \leftarrow r$



Key Idea #2: The Minimized $L(n_1)$ Can be "Absorbed" by Changing the Order of Evaluation

Instructions

$r \leftarrow r \ op \ m$
$r_1 \leftarrow r_1 \ op \ r_2$
$r \leftarrow m$
$m \leftarrow r$



$r_0 \leftarrow a$
$r_0 \leftarrow r_0 * b$

$r_1 \leftarrow c$
$r_2 \leftarrow d$
$r_2 \leftarrow r_2 - e$
$r_1 \leftarrow r_1 / r_2$

$r_0 \leftarrow r_0 + r_1$

$r_1 \leftarrow c$
$r_0 \leftarrow d$
$r_0 \leftarrow r_0 - e$
$r_1 \leftarrow r_1 / r_0$

$r_0 \leftarrow a$
$r_0 \leftarrow r_0 * b$

$r_0 \leftarrow r_0 + r_1$

$r_0 \leftarrow d$
$r_0 \leftarrow r_0 - e$
$r_1 \leftarrow c$
$r_1 \leftarrow r_1 / r_0$

$r_0 \leftarrow a$
$r_0 \leftarrow r_0 * b$

$r_0 \leftarrow r_0 + r_1$

IIT Bombay
cs302: Implementation of Programming Languages

Topic:

Code Generation

Section:

Global Register Allocation

Managing Registers Across Calls

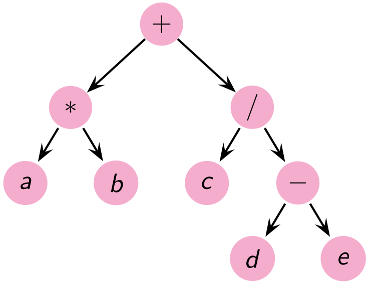
Registers Usage in sclp

Instruction Selection

Key Idea #2: The Minimized $L(n_1)$ Can be “Absorbed” by Changing the Order of Evaluation

Instructions

$r \leftarrow r \; op \; m$
$r_1 \leftarrow r_1 \; op \; r_2$
$r \leftarrow m$
$m \leftarrow r$

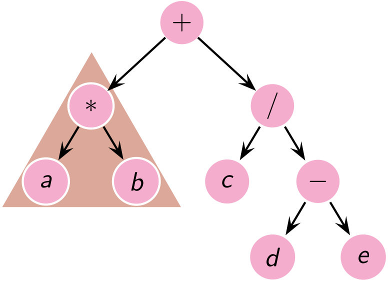


Program 1 Regs r_0, r_1, r_2	Program 2 Regs r_0, r_1	Program 3 Regs r_0, r_1
$r_0 \leftarrow a$	$r_1 \leftarrow c$	$r_0 \leftarrow d$
$r_0 \leftarrow r_0 * b$	$r_0 \leftarrow d$	$r_0 \leftarrow r_0 - e$
$r_1 \leftarrow c$	$r_0 \leftarrow r_0 - e$	$r_1 \leftarrow c$
$r_2 \leftarrow d$	$r_1 \leftarrow r_1 / r_0$	$r_1 \leftarrow r_1 / r_0$
$r_2 \leftarrow r_2 - e$	$r_0 \leftarrow a$	$r_0 \leftarrow a$
$r_1 \leftarrow r_1 / r_2$	$r_0 \leftarrow r_0 * b$	$r_0 \leftarrow r_0 * b$
$r_0 \leftarrow r_0 + r_1$	$r_0 \leftarrow r_0 + r_1$	$r_0 \leftarrow r_0 + r_1$

Key Idea #2: The Minimized $L(n_1)$ Can be "Absorbed" by Changing the Order of Evaluation

Instructions

$r \leftarrow r \ op\ m$
$r_1 \leftarrow r_1 \ op\ r_2$
$r \leftarrow m$
$m \leftarrow r$



Program 1
Regs r_0, r_1, r_2

$r_0 \leftarrow a$
$r_0 \leftarrow r_0 * b$

$r_1 \leftarrow c$
$r_2 \leftarrow d$
$r_2 \leftarrow r_2 - e$
$r_1 \leftarrow r_1 / r_2$
$r_0 \leftarrow r_0 + r_1$

Program 2
Regs r_0, r_1

$r_1 \leftarrow c$
$r_0 \leftarrow d$
$r_0 \leftarrow r_0 - e$
$r_1 \leftarrow r_1 / r_0$

$r_0 \leftarrow a$
$r_0 \leftarrow r_0 * b$
$r_0 \leftarrow r_0 + r_1$

Program 3
Regs r_0, r_1

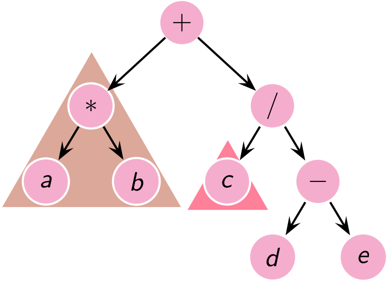
$r_0 \leftarrow d$
$r_0 \leftarrow r_0 - e$
$r_1 \leftarrow c$
$r_1 \leftarrow r_1 / r_0$

$r_0 \leftarrow a$
$r_0 \leftarrow r_0 * b$
$r_0 \leftarrow r_0 + r_1$

Key Idea #2: The Minimized $L(n_1)$ Can be "Absorbed" by Changing the Order of Evaluation

Instructions

$r \leftarrow r \; op \; m$
$r_1 \leftarrow r_1 \; op \; r_2$
$r \leftarrow m$
$m \leftarrow r$



Program 1
Regs r_0, r_1, r_2

$r_0 \leftarrow a$
$r_0 \leftarrow r_0 * b$

$r_1 \leftarrow c$
$r_2 \leftarrow d$
$r_2 \leftarrow r_2 - e$
$r_1 \leftarrow r_1 / r_2$

$r_0 \leftarrow r_0 + r_1$

Program 2
Regs r_0, r_1

$r_1 \leftarrow c$
$r_0 \leftarrow d$
$r_0 \leftarrow r_0 - e$
$r_1 \leftarrow r_1 / r_0$

$r_0 \leftarrow a$
$r_0 \leftarrow r_0 * b$

$r_0 \leftarrow r_0 + r_1$

Program 3
Regs r_0, r_1

$r_0 \leftarrow d$
$r_0 \leftarrow r_0 - e$
$r_1 \leftarrow c$
$r_1 \leftarrow r_1 / r_0$

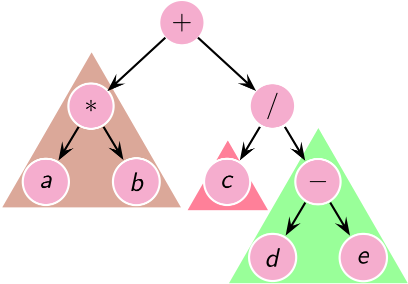
$r_0 \leftarrow a$
$r_0 \leftarrow r_0 * b$

$r_0 \leftarrow r_0 + r_1$

Key Idea #2: The Minimized $L(n_1)$ Can be "Absorbed" by Changing the Order of Evaluation

Instructions

$r \leftarrow r \; op \; m$
$r_1 \leftarrow r_1 \; op \; r_2$
$r \leftarrow m$
$m \leftarrow r$



Program 1
Regs r_0, r_1, r_2

$r_0 \leftarrow a$
$r_0 \leftarrow r_0 * b$
$r_1 \leftarrow c$
$r_2 \leftarrow d$
$r_2 \leftarrow r_2 - e$
$r_1 \leftarrow r_1 / r_2$
$r_0 \leftarrow r_0 + r_1$

Program 2
Regs r_0, r_1

$r_1 \leftarrow c$
$r_0 \leftarrow d$
$r_0 \leftarrow r_0 - e$
$r_1 \leftarrow r_1 / r_0$
$r_0 \leftarrow a$
$r_0 \leftarrow r_0 * b$
$r_0 \leftarrow r_0 + r_1$

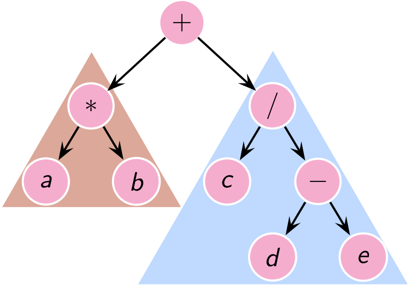
Program 3
Regs r_0, r_1

$r_0 \leftarrow d$
$r_0 \leftarrow r_0 - e$
$r_1 \leftarrow c$
$r_1 \leftarrow r_1 / r_0$
$r_0 \leftarrow a$
$r_0 \leftarrow r_0 * b$
$r_0 \leftarrow r_0 + r_1$

Key Idea #2: The Minimized $L(n_1)$ Can be "Absorbed" by Changing the Order of Evaluation

Instructions

$r \leftarrow r\ op\ m$
$r_1 \leftarrow r_1\ op\ r_2$
$r \leftarrow m$
$m \leftarrow r$

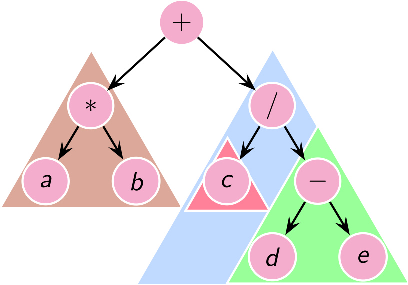


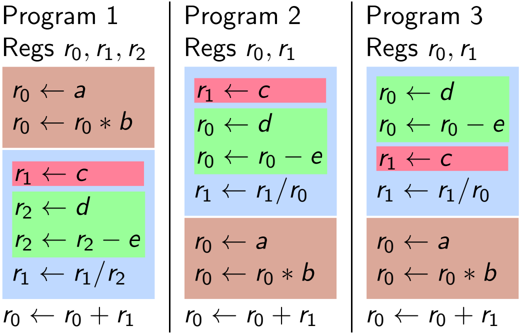
Program 1	Program 2	Program 3
Regs r_0, r_1, r_2	Regs r_0, r_1	Regs r_0, r_1
$r_0 \leftarrow a$ $r_0 \leftarrow r_0 * b$	$r_1 \leftarrow c$ $r_0 \leftarrow d$ $r_0 \leftarrow r_0 - e$ $r_1 \leftarrow r_1 / r_0$	$r_0 \leftarrow d$ $r_0 \leftarrow r_0 - e$ $r_1 \leftarrow c$ $r_1 \leftarrow r_1 / r_0$
$r_1 \leftarrow c$ $r_2 \leftarrow d$ $r_2 \leftarrow r_2 - e$ $r_1 \leftarrow r_1 / r_2$	$r_0 \leftarrow a$ $r_0 \leftarrow r_0 * b$	$r_0 \leftarrow a$ $r_0 \leftarrow r_0 * b$
$r_0 \leftarrow r_0 + r_1$	$r_0 \leftarrow r_0 + r_1$	$r_0 \leftarrow r_0 + r_1$

Key Idea #2: The Minimized $L(n_1)$ Can be "Absorbed" by Changing the Order of Evaluation

Instructions

$r \leftarrow r \ op \ m$
$r_1 \leftarrow r_1 \ op \ r_2$
$r \leftarrow m$
$m \leftarrow r$





Program 1
Regs r_0, r_1, r_2

$r_0 \leftarrow a$
$r_0 \leftarrow r_0 * b$
$r_1 \leftarrow c$
$r_2 \leftarrow d$
$r_2 \leftarrow r_2 - e$
$r_1 \leftarrow r_1 / r_2$
$r_0 \leftarrow r_0 + r_1$

Program 2
Regs r_0, r_1

$r_1 \leftarrow c$
$r_0 \leftarrow d$
$r_0 \leftarrow r_0 - e$
$r_1 \leftarrow r_1 / r_0$
$r_0 \leftarrow a$
$r_0 \leftarrow r_0 * b$
$r_0 \leftarrow r_0 + r_1$

Program 3
Regs r_0, r_1

$r_0 \leftarrow d$
$r_0 \leftarrow r_0 - e$
$r_1 \leftarrow c$
$r_1 \leftarrow r_1 / r_0$
$r_0 \leftarrow a$
$r_0 \leftarrow r_0 * b$
$r_0 \leftarrow r_0 + r_1$

The ordering between the pink and green subtrees does not matter (programs 2 and 3)

The ordering between the brown and lightblue subtrees affects the number of registers

We want to minimise the number of registers so that we do not need to store an intermediate result in memory

The Sethi-Ullman Algorithm

- Traverse the expression tree bottom up and label each node with the minimum number of registers needed to evaluate the subexpression rooted at the node
- Traverse the expression tree top down and generate code

The Sethi-Ullman Algorithm

- Traverse the expression tree bottom up and label each node with the minimum number of registers needed to evaluate the subexpression rooted at the node

  Each node is processed exactly once

- Traverse the expression tree top down and generate code

  Each node is processed exactly once

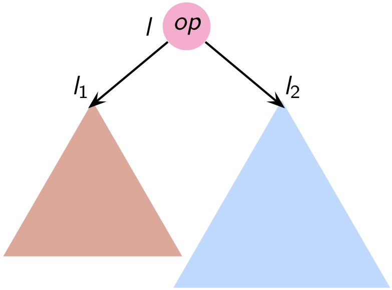
The Sethi-Ullman Algorithm

- Traverse the expression tree bottom up and label each node with the minimum number of registers needed to evaluate the subexpression rooted at the node

  Each node is processed exactly once
- Traverse the expression tree top down and generate code

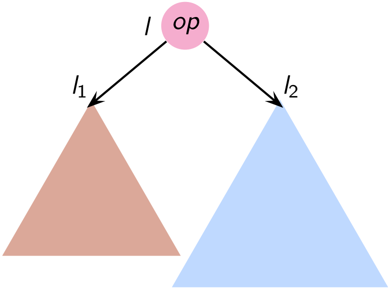
  Each node is processed exactly once
- The algorithm is linear in the size of the expression tree

Labelling the Expression Tree

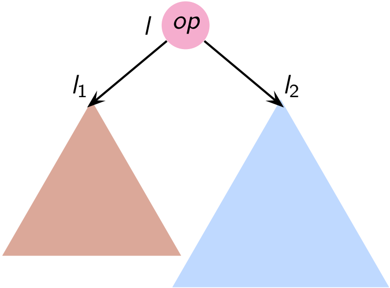


Labelling the Expression Tree

Assume that the register requirements of the two subtrees are l_1 and l_2 and that $l_1 < l_2$



Labelling the Expression Tree

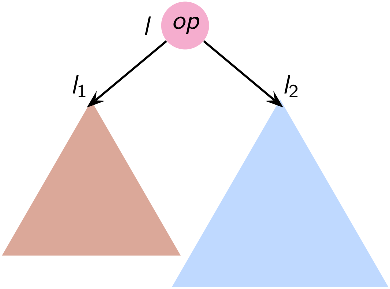


Assume that the register requirements of the two subtrees are l_1 and l_2 and that $l_1 < l_2$

- If we evaluate the brown subtree first, we need l_1 registers to evaluate it, 1 register to hold its result and l_2 registers to evaluate the blue subtree

$$l = max(l_1, l_2 + 1) = l_2 + 1$$

Labelling the Expression Tree



Assume that the register requirements of the two subtrees are l_1 and l_2 and that $l_1 < l_2$

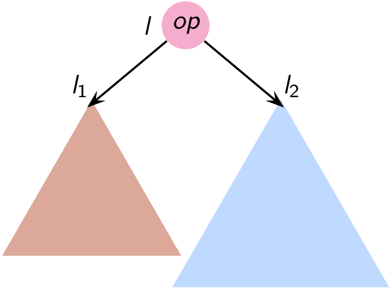
- If we evaluate the brown subtree first, we need l_1 registers to evaluate it, 1 register to hold its result and l_2 registers to evaluate the blue subtree

$$l = max(l_1, l_2 + 1) = l_2 + 1$$

- If we evaluate the blue subtree first, we need l_2 registers to evaluate it, 1 register to hold its result and l_1 registers to evaluate the brown subtree

$$l = max(l_1 + 1, l_2) = l_2$$

Labelling the Expression Tree



Assume that the register requirements of the two subtrees are l_1 and l_2 and that $l_1 < l_2$

- If we evaluate the brown subtree first, we need l_1 registers to evaluate it, 1 register to hold its result and l_2 registers to evaluate the blue subtree

$$l = max(l_1, l_2 + 1) = l_2 + 1$$

- If we evaluate the blue subtree first, we need l_2 registers to evaluate it, 1 register to hold its result and l_1 registers to evaluate the brown subtree

$$l = max(l_1 + 1, l_2) = l_2$$

Evaluate the subtree with larger requirements first

Labelling the Expression Tree

- For instruction with arity-2 and binary tree

$$label(n) = \begin{cases} 1 & n \text{ is a leaf and must be in a register} \\ 0 & n \text{ is a leaf and can be in memory} \\ \max(label(n_1), label(n_2)) & \begin{array}{l} n \text{ has two child nodes } n_1 \text{ and } n_2 \\ \text{and } label(n_1) \neq label(n_2) \end{array} \\ label(n_1) + 1 & \begin{array}{l} n \text{ has two child nodes } n_1 \text{ and } n_2 \\ \text{and } label(n_1) = label(n_2) \end{array} \end{cases}$$

Labelling the Expression Tree

- For instruction with arity-2 and binary tree

$$label(n) = \begin{cases} 1 & n \text{ is a leaf and must be in a register} \\ 0 & n \text{ is a leaf and can be in memory} \\ \max(label(n_1), label(n_2)) & \begin{array}{l} n \text{ has two child nodes } n_1 \text{ and } n_2 \\ \text{and } label(n_1) \neq label(n_2) \end{array} \\ label(n_1) + 1 & \begin{array}{l} n \text{ has two child nodes } n_1 \text{ and } n_2 \\ \text{and } label(n_1) = label(n_2) \end{array} \end{cases}$$

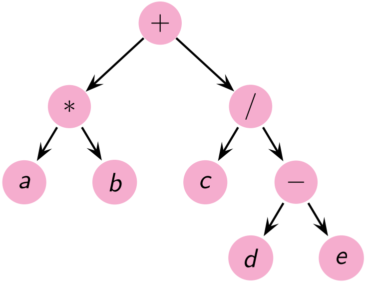
- Generalizing to instructions and trees of higher arity
  Let node n have k children with the labels $l_1 \geq l_2 \geq \ldots \geq l_k$

$$label(n) = \max(l_j + j - 1), 1 \leq j \leq k$$

Labelling the Expression Tree

Instructions

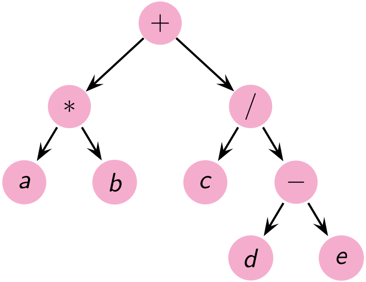
$r \leftarrow r\ op\ m$
$r_1 \leftarrow r_1\ op\ r_2$
$r \leftarrow m$
$m \leftarrow r$



Labelling the Expression Tree

Instructions

$r \leftarrow r\ op\ m$
$r_1 \leftarrow r_1\ op\ r_2$
$r \leftarrow m$
$m \leftarrow r$

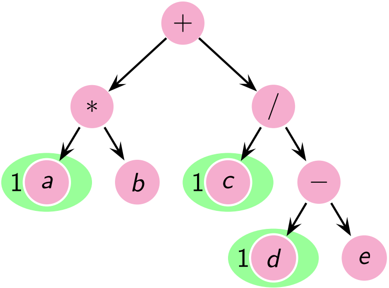


$$l_n = \begin{cases} 1 & n \text{ is a left leaf} \\ 0 & n \text{ is a right leaf} \\ \max(l_1, l_2) & n \text{ has two children with labels } l_1 \neq l_2 \\ l_1 + 1 & n \text{ has two children with labels } l_1 = l_2 \end{cases}$$

Labelling the Expression Tree

Instructions

$r \leftarrow r\ op\ m$
$r_1 \leftarrow r_1\ op\ r_2$
$r \leftarrow m$
$m \leftarrow r$



$$l_n = \begin{cases} 1 & n \text{ is a left leaf} \\ 0 & n \text{ is a right leaf} \\ \max(l_1, l_2) & n \text{ has two children with labels } l_1 \neq l_2 \\ l_1 + 1 & n \text{ has two children with labels } l_1 = l_2 \end{cases}$$

Labelling the Expression Tree

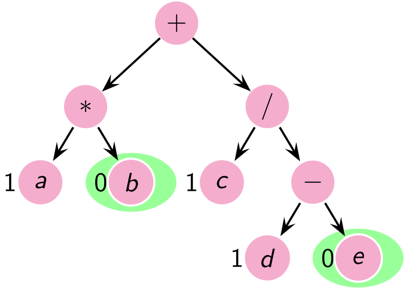
Instructions

$r \leftarrow r\ op\ m$

$r_1 \leftarrow r_1\ op\ r_2$

$r \leftarrow m$

$m \leftarrow r$

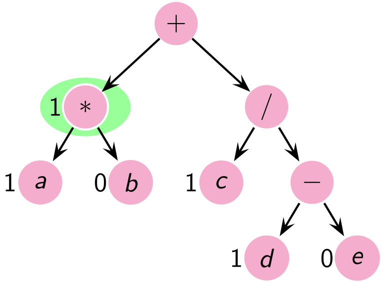


$$l_n = \begin{cases} 1 & n \text{ is a left leaf} \\ 0 & n \text{ is a right leaf} \\ \max(l_1, l_2) & n \text{ has two children with labels } l_1 \neq l_2 \\ l_1 + 1 & n \text{ has two children with labels } l_1 = l_2 \end{cases}$$

Labelling the Expression Tree

Instructions

$r \leftarrow r \; op \; m$
$r_1 \leftarrow r_1 \; op \; r_2$
$r \leftarrow m$
$m \leftarrow r$

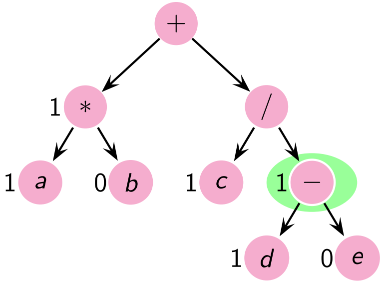


$$l_n = \begin{cases} 1 & n \text{ is a left leaf} \\ 0 & n \text{ is a right leaf} \\ \max(l_1, l_2) & n \text{ has two children with labels } l_1 \neq l_2 \\ l_1 + 1 & n \text{ has two children with labels } l_1 = l_2 \end{cases}$$

Labelling the Expression Tree

Instructions

$r \leftarrow r \; op \; m$
$r_1 \leftarrow r_1 \; op \; r_2$
$r \leftarrow m$
$m \leftarrow r$

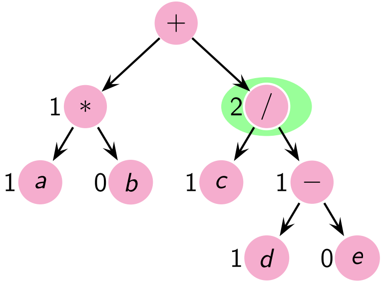


$$l_n = \begin{cases} 1 & n \text{ is a left leaf} \\ 0 & n \text{ is a right leaf} \\ \max(l_1, l_2) & n \text{ has two children with labels } l_1 \neq l_2 \\ l_1 + 1 & n \text{ has two children with labels } l_1 = l_2 \end{cases}$$

Labelling the Expression Tree

Instructions

$r \leftarrow r \; op \; m$
$r_1 \leftarrow r_1 \; op \; r_2$
$r \leftarrow m$
$m \leftarrow r$

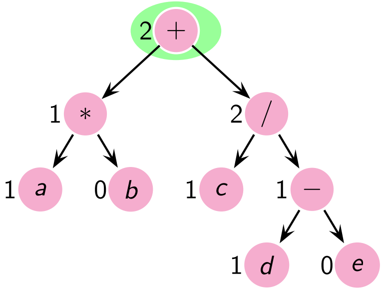


$$l_n = \begin{cases} 1 & n \text{ is a left leaf} \\ 0 & n \text{ is a right leaf} \\ \max(l_1, l_2) & n \text{ has two children with labels } l_1 \neq l_2 \\ l_1 + 1 & n \text{ has two children with labels } l_1 = l_2 \end{cases}$$

Labelling the Expression Tree

Instructions

$r \leftarrow r \; op \; m$
$r_1 \leftarrow r_1 \; op \; r_2$
$r \leftarrow m$
$m \leftarrow r$

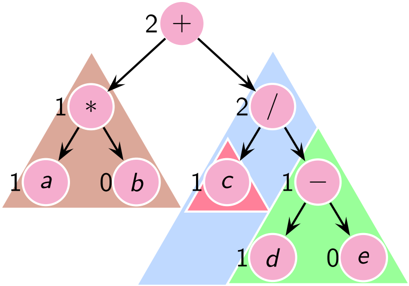


$$l_n = \begin{cases} 1 & n \text{ is a left leaf} \\ 0 & n \text{ is a right leaf} \\ \max(l_1, l_2) & n \text{ has two children with labels } l_1 \neq l_2 \\ l_1 + 1 & n \text{ has two children with labels } l_1 = l_2 \end{cases}$$

Labelling the Expression Tree

Instructions

$r \leftarrow r \; op \; m$
$r_1 \leftarrow r_1 \; op \; r_2$
$r \leftarrow m$
$m \leftarrow r$



Program 1
Regs r_0, r_1, r_2

$r_0 \leftarrow a$
$r_0 \leftarrow r_0 * b$
$r_1 \leftarrow c$
$r_2 \leftarrow d$
$r_2 \leftarrow r_2 - e$
$r_1 \leftarrow r_1 / r_2$
$r_0 \leftarrow r_0 + r_1$

Program 2
Regs r_0, r_1

$r_1 \leftarrow c$
$r_0 \leftarrow d$
$r_0 \leftarrow r_0 - e$
$r_1 \leftarrow r_1 / r_0$
$r_0 \leftarrow a$
$r_0 \leftarrow r_0 * b$
$r_0 \leftarrow r_0 + r_1$

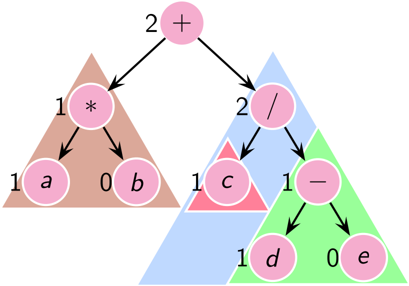
Program 3
Regs r_0, r_1

$r_0 \leftarrow d$
$r_0 \leftarrow r_0 - e$
$r_1 \leftarrow c$
$r_1 \leftarrow r_1 / r_0$
$r_0 \leftarrow a$
$r_0 \leftarrow r_0 * b$
$r_0 \leftarrow r_0 + r_1$

Labelling the Expression Tree

Instructions

$r \leftarrow r \ op \ m$
$r_1 \leftarrow r_1 \ op \ r_2$
$r \leftarrow m$
$m \leftarrow r$



Program 1	Program 2	Program 3
Regs r_0, r_1, r_2	Regs r_0, r_1	Regs r_0, r_1
$r_0 \leftarrow a$	$r_1 \leftarrow c$	$r_0 \leftarrow d$
$r_0 \leftarrow r_0 * b$	$r_0 \leftarrow d$	$r_0 \leftarrow r_0 - e$
$r_1 \leftarrow c$	$r_0 \leftarrow r_0 - e$	$r_1 \leftarrow c$
$r_2 \leftarrow d$	$r_1 \leftarrow r_1 / r_0$	$r_1 \leftarrow r_1 / r_0$
$r_2 \leftarrow r_2 - e$	$r_0 \leftarrow a$	$r_0 \leftarrow a$
$r_1 \leftarrow r_1 / r_2$	$r_0 \leftarrow r_0 * b$	$r_0 \leftarrow r_0 * b$
$r_0 \leftarrow r_0 + r_1$	$r_0 \leftarrow r_0 + r_1$	$r_0 \leftarrow r_0 + r_1$

Program 1 computes the brown subtree first whereas programs 2 and 3 compute the blue subtree before the brown subtree; hence program 1 needs one register more than programs 2 and 3

The pink and the green subtrees have the same labels and the order does not matter; hence programs 2 and 3 have the same register requirements for the blue subtree

Generating Code for a Labelled Tree

- Recursive procedure $gencode(n)$ to generate code for node n
    - $rstack$ is a stack of registers; $gencode(n)$ generates the code such that the result of the subtree rooted at n is contained in $top(rstack)$
    - $tstack$ is a stack of temporaries used when the algorithm runs out of registers
    - $swap(rstack)$ swaps the two top registers in $rstack$
    - Procedure $emit$ emits a single statement of the generated code

Generating Code for a Labelled Tree with k Registers

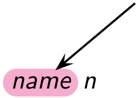
- Procedure *gencode*(n) needs to consider the following five cases
    1. n is a left leaf
    2. The right child of n is a leaf
    3. label(left child) $\geq$ label(right child) and label(right child) $< k$
       (label(left child) can be $\geq k$)
    4. label(left child) $<$ label(right child) and label(left child) $< k$
       (label(right child) can be $\geq k$)
    5. label(left child) $\geq k$ and label(right child) $\geq k$

Generating Code for a Labelled Tree with k Registers

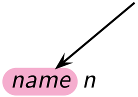
- Procedure *gencode*(n) needs to consider the following five cases
  1. n is a left leaf
  2. The right child of n is a leaf
  3. label(left child) $\geq$ label(right child) and label(right child) $< k$
     (label(left child) can be $\geq k$)
  4. label(left child) $<$ label(right child) and label(left child) $< k$
     (label(right child) can be $\geq k$)
  5. label(left child) $\geq k$ and label(right child) $\geq k$
- The above cases are exhaustive
  - Cases 1 and 2. Number k is irrelevant
  - Cases 2 and 3. At least one subtree has a label smaller than k
  - Case 5. The labels of both subtrees is greater than or equal to k

Generating Code for a Labelled Tree: Case 1

Node n is a left leaf



Generating Code for a Labelled Tree: Case 1

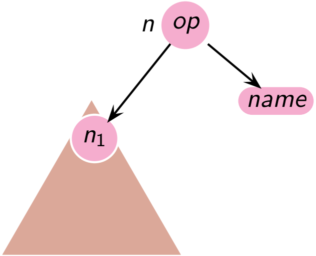


Node n is a left leaf

Procedure $gencode(n)$ is

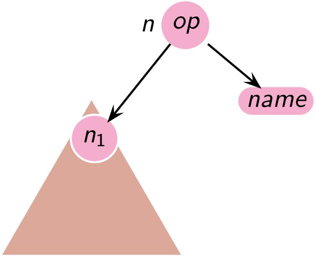
$$\boxed{emit(top(rstack) \leftarrow name)}$$

Generating Code for a Labelled Tree: Case 2



The right child of n is a leaf

Generating Code for a Labelled Tree: Case 2



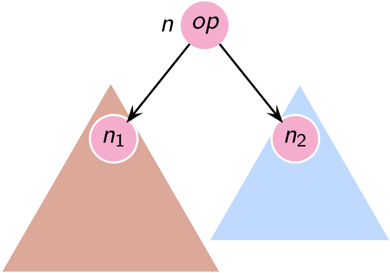
The right child of n is a leaf

Procedure $gencode(n)$ is

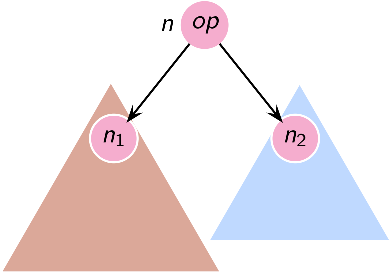
```
$gencode(n_1)$
$emit(top(rstack) \leftarrow top(rstack)\ op\ name)$
```

Generating Code for a Labelled Tree: Case 3

$label(n_1) \geq label(n_2)$ and $label(n_2) < k$



Generating Code for a Labelled Tree: Case 3



$label(n_1) \geq label(n_2)$ and $label(n_2) < k$

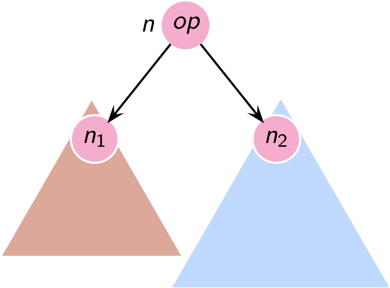
Procedure $gencode(n)$ is

```
$gencode(n_1)$
$R = pop(rstack)$
$gencode(n_2)$
$emit(R \leftarrow R\ op\ top(rstack))$
$push(R, rstack)$
```

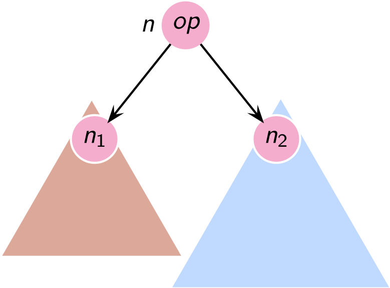
Contiguous evaluation guarantees that evaluation of n_2 is denied only a single register R

Generating Code for a Labelled Tree: Case 4

$label(n_1) < label(n_2)$ and $label(n_1) < k$



Generating Code for a Labelled Tree: Case 4



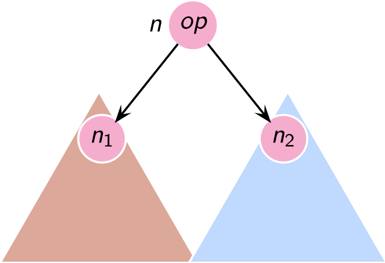
$label(n_1) < label(n_2)$ and $label(n_1) < k$

Procedure $gencode(n)$ is

```
$swap(rstack)$
$gencode(n_2)$
$R = pop(rstack)$
$gencode(n_1)$
$emit(top(rstack) \leftarrow top(rstack)\ op\ R)$
$push(R, rstack)$
$swap(rstack)$
```

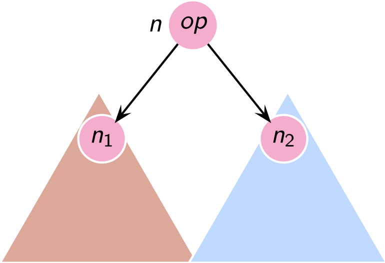
Contiguous evaluation guarantees that evaluation of n_1 is denied only a single register R

Generating Code for a Labelled Tree: Case 5



$label(n_1) \geq k$ and $label(n_2) \geq k$

Generating Code for a Labelled Tree: Case 5

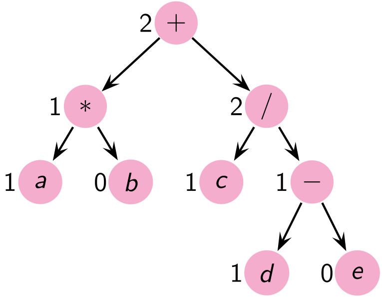


$label(n_1) \geq k$ and $label(n_2) \geq k$

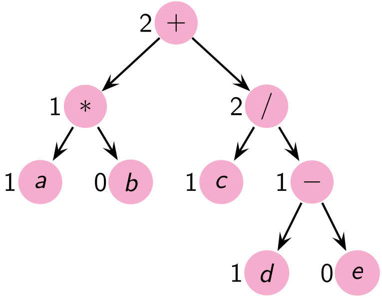
Procedure $gencode(n)$ is

```
$gencode(n_2)$
$T = pop(tstack)$
$emit(T \leftarrow top(rstack))$
$gencode(n_1)$
$emit(top(rstack) \leftarrow top(rstack)\ op\ T)$
$push(T, tstack)$
```

Code Generation with Two Registers r_0 and r_1



Code Generation with Two Registers r_0 and r_1



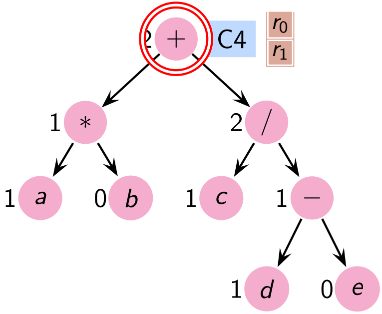
We highlight the nodes in the expression tree

We show the control stack of invocations of procedure *gencode* and highlight the statements for cases 3 and 4

The red font shows the statements that are being executed whereas blue font shows the statements whose execution is over

C1	$emit(top(rstack) \leftarrow name)$	
C2	$gencode(n_1)$ $emit(top(rstack) \leftarrow top(rstack)\ op\ name)$	
C3	$gencode(n_1)$ $R = pop(rstack)$ $gencode(n_2)$ $emit(R \leftarrow R\ op\ top(rstack))$ $push(R, rstack)$	label(right) $<$ k label(left) $\geq$ label(right)
C4	$swap(rstack)$ $gencode(n_2)$ $R = pop(rstack)$ $gencode(n_1)$ $emit(top(rstack) \leftarrow top(rstack)\ op\ R)$ $push(R, rstack)$ $swap(rstack)$	label(left) $<$ k label(right) $>$ label(left)
C5	$gencode(n_2)$ $T = pop(tstack)$ $emit(T \leftarrow top(rstack))$ $gencode(n_1)$ $emit(top(rstack) \leftarrow top(rstack)\ op\ T)$ $push(T, tstack)$	

Code Generation with Two Registers r_0 and r_1

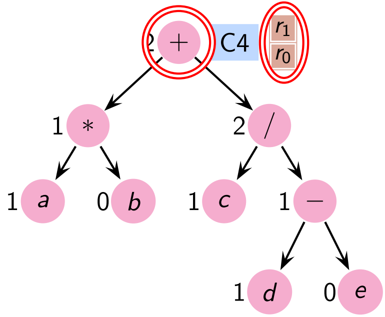


$g(+) : C4$

Control Stack

C1	$emit(top(rstack) \leftarrow name)$	
C2	$gencode(n_1)$ $emit(top(rstack) \leftarrow top(rstack)\ op\ name)$	
C3	$gencode(n_1)$ $R = pop(rstack)$ $gencode(n_2)$ $emit(R \leftarrow R\ op\ top(rstack))$ $push(R, rstack)$	label(right) $<$ k label(left) $\geq$ label(right)
C4	$swap(rstack)$ $gencode(n_2)$ $R = pop(rstack)$ $gencode(n_1)$ $emit(top(rstack) \leftarrow top(rstack)\ op\ R)$ $push(R, rstack)$ $swap(rstack)$	label(left) $<$ k label(right) $>$ label(left)
C5	$gencode(n_2)$ $T = pop(tstack)$ $emit(T \leftarrow top(rstack))$ $gencode(n_1)$ $emit(top(rstack) \leftarrow top(rstack)\ op\ T)$ $push(T, tstack)$	

Code Generation with Two Registers r_0 and r_1

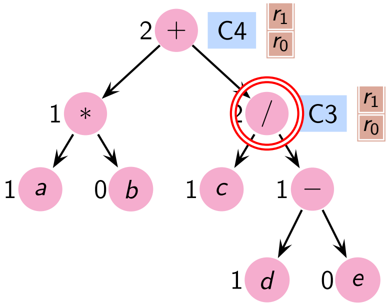


$g(+) : C4$

Control Stack

C1	*emit(top(rstack)* $\leftarrow$ *name)*	
C2	*gencode(n_1)* *emit(top(rstack)* $\leftarrow$ *top(rstack) op name)*	
C3	*gencode(n_1)* *R = pop(rstack)* *gencode(n_2)* *emit(R* $\leftarrow$ *R op top(rstack))* *push(R, rstack)*	label(right) $<$ k label(left) $\geq$ label(right)
C4	*swap(rstack)* *gencode(n_2)* *R = pop(rstack)* *gencode(n_1)* *emit(top(rstack)* $\leftarrow$ *top(rstack) op R)* *push(R, rstack)* *swap(rstack)*	label(left) $<$ k label(right) $>$ label(left)
C5	*gencode(n_2)* *T = pop(tstack)* *emit(T* $\leftarrow$ *top(rstack))* *gencode(n_1)* *emit(top(rstack)* $\leftarrow$ *top(rstack) op T)* *push(T, tstack)*	

Code Generation with Two Registers r_0 and r_1

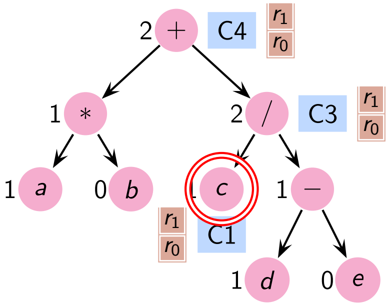


$g(/) : C3$
$g(+) : C4$

Control Stack

C1	$emit(top(rstack) \leftarrow name)$	
C2	$gencode(n_1)$ $emit(top(rstack) \leftarrow top(rstack)\ op\ name)$	
C3	$gencode(n_1)$ $R = pop(rstack)$ $gencode(n_2)$ $emit(R \leftarrow R\ op\ top(rstack))$ $push(R, rstack)$	label(right) $<$ k label(left) $\geq$ label(right)
C4	$swap(rstack)$ $gencode(n_2)$ $R = pop(rstack)$ $gencode(n_1)$ $emit(top(rstack) \leftarrow top(rstack)\ op\ R)$ $push(R, rstack)$ $swap(rstack)$	label(left) $<$ k label(right) $>$ label(left)
C5	$gencode(n_2)$ $T = pop(tstack)$ $emit(T \leftarrow top(rstack))$ $gencode(n_1)$ $emit(top(rstack) \leftarrow top(rstack)\ op\ T)$ $push(T, tstack)$	

Code Generation with Two Registers r_0 and r_1



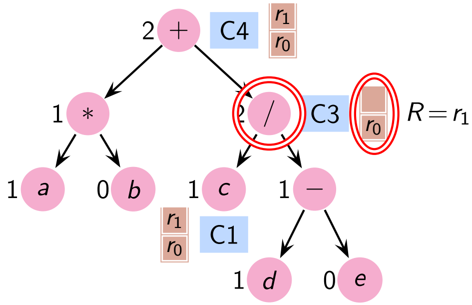
$r_1 \leftarrow c$

$g(c) : C1$
$g(/) : C3$
$g(+) : C4$

Control Stack

C1	$emit(top(rstack) \leftarrow name)$	
C2	$gencode(n_1)$ $emit(top(rstack) \leftarrow top(rstack)\ op\ name)$	
C3	$gencode(n_1)$ $R = pop(rstack)$ $gencode(n_2)$ $emit(R \leftarrow R\ op\ top(rstack))$ $push(R, rstack)$	label(right) $<$ k label(left) $\geq$ label(right)
C4	$swap(rstack)$ $gencode(n_2)$ $R = pop(rstack)$ $gencode(n_1)$ $emit(top(rstack) \leftarrow top(rstack)\ op\ R)$ $push(R, rstack)$ $swap(rstack)$	label(left) $<$ k label(right) $>$ label(left)
C5	$gencode(n_2)$ $T = pop(tstack)$ $emit(T \leftarrow top(rstack))$ $gencode(n_1)$ $emit(top(rstack) \leftarrow top(rstack)\ op\ T)$ $push(T, tstack)$	

Code Generation with Two Registers r_0 and r_1



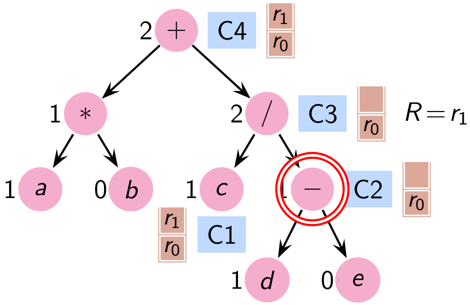
$r_1 \leftarrow c$

$g(/) : C3$
$g(+) : C4$

Control Stack

C1	$emit(top(rstack) \leftarrow name)$	
C2	$gencode(n_1)$ $emit(top(rstack) \leftarrow top(rstack)\ op\ name)$	
C3	$gencode(n_1)$ $R = pop(rstack)$ $gencode(n_2)$ $emit(R \leftarrow R\ op\ top(rstack))$ $push(R, rstack)$	label(right) $<$ k label(left) $\geq$ label(right)
C4	$swap(rstack)$ $gencode(n_2)$ $R = pop(rstack)$ $gencode(n_1)$ $emit(top(rstack) \leftarrow top(rstack)\ op\ R)$ $push(R, rstack)$ $swap(rstack)$	label(left) $<$ k label(right) $>$ label(left)
C5	$gencode(n_2)$ $T = pop(tstack)$ $emit(T \leftarrow top(rstack))$ $gencode(n_1)$ $emit(top(rstack) \leftarrow top(rstack)\ op\ T)$ $push(T, tstack)$	

Code Generation with Two Registers r_0 and r_1



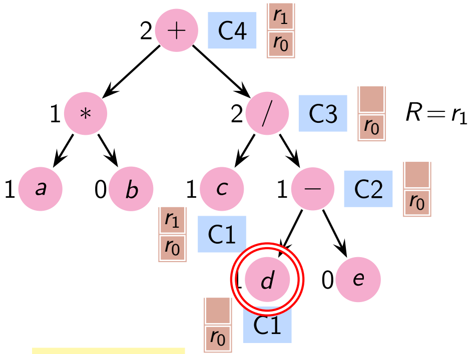
$r_1 \leftarrow c$

$g(-) : C2$
$g(/) : C3$
$g(+) : C4$

Control Stack

```
C1  emit(top(rstack) ← name)

C2  gencode(n1)
    emit(top(rstack) ← top(rstack) op name)

C3  gencode(n1)                           label(right) < k
    R = pop(rstack)                       label(left) ≥ label(right)
    gencode(n2)
    emit(R ← R op top(rstack))
    push(R, rstack)

C4  swap(rstack)                          label(left) < k
    gencode(n2)                           label(right) > label(left)
    R = pop(rstack)
    gencode(n1)
    emit(top(rstack) ← top(rstack) op R)
    push(R, rstack)
    swap(rstack)

C5  gencode(n2)
    T = pop(tstack)
    emit(T ← top(rstack))
    gencode(n1)
    emit(top(rstack) ← top(rstack) op T)
    push(T, tstack)
```

Code Generation with Two Registers r_0 and r_1



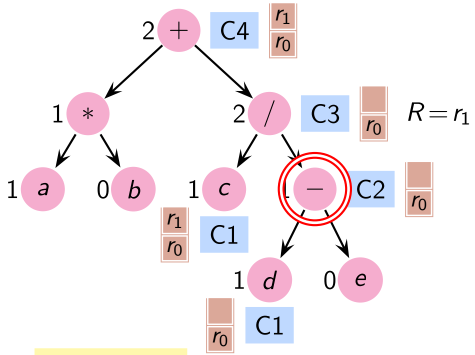
$r_1 \leftarrow c$
$r_0 \leftarrow d$

$g(d) : C1$
$g(-) : C2$
$g(/) : C3$
$g(+) : C4$

Control Stack

C1	*emit*(*top*(*rstack*) ← *name*)	
C2	*gencode*(n_1) *emit*(*top*(*rstack*) ← *top*(*rstack*) *op name*)	
C3	*gencode*(n_1) *R* = *pop*(*rstack*) *gencode*(n_2) *emit*(*R* ← *R op top*(*rstack*)) *push*(*R*, *rstack*)	label(right) < k label(left) ≥ label(right)
C4	*swap*(*rstack*) *gencode*(n_2) *R* = *pop*(*rstack*) *gencode*(n_1) *emit*(*top*(*rstack*) ← *top*(*rstack*) *op R*) *push*(*R*, *rstack*) *swap*(*rstack*)	label(left) < k label(right) > label(left)
C5	*gencode*(n_2) *T* = *pop*(*tstack*) *emit*(*T* ← *top*(*rstack*)) *gencode*(n_1) *emit*(*top*(*rstack*) ← *top*(*rstack*) *op T*) *push*(*T*, *tstack*)	

Code Generation with Two Registers r_0 and r_1



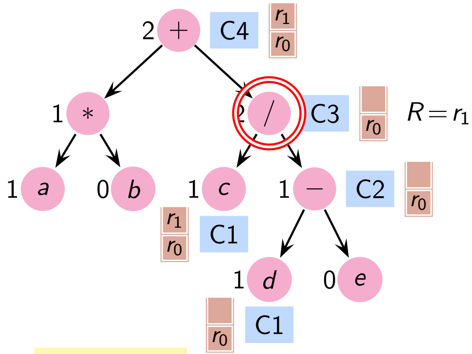
$r_1 \leftarrow c$
$r_0 \leftarrow d$
$r_0 \leftarrow r_0 - e$

$g(-) : C2$
$g(/) : C3$
$g(+) : C4$

Control Stack

C1	$emit(top(rstack) \leftarrow name)$	
C2	$gencode(n_1)$ $emit(top(rstack) \leftarrow top(rstack)\ op\ name)$	
C3	$gencode(n_1)$ $R = pop(rstack)$ $gencode(n_2)$ $emit(R \leftarrow R\ op\ top(rstack))$ $push(R, rstack)$	label(right) $<$ k label(left) $\geq$ label(right)
C4	$swap(rstack)$ $gencode(n_2)$ $R = pop(rstack)$ $gencode(n_1)$ $emit(top(rstack) \leftarrow top(rstack)\ op\ R)$ $push(R, rstack)$ $swap(rstack)$	label(left) $<$ k label(right) $>$ label(left)
C5	$gencode(n_2)$ $T = pop(tstack)$ $emit(T \leftarrow top(rstack))$ $gencode(n_1)$ $emit(top(rstack) \leftarrow top(rstack)\ op\ T)$ $push(T, tstack)$	

Code Generation with Two Registers r_0 and r_1



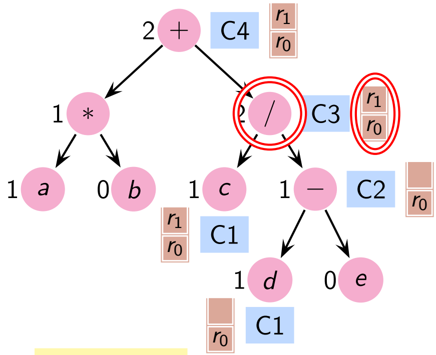
$r_1 \leftarrow c$
$r_0 \leftarrow d$
$r_0 \leftarrow r_0 - e$
$r_1 \leftarrow r_1 / r_0$

$g(/) : C3$
$g(+) : C4$

Control Stack

C1	$emit(top(rstack) \leftarrow name)$	
C2	$gencode(n_1)$ $emit(top(rstack) \leftarrow top(rstack)\ op\ name)$	
C3	$gencode(n_1)$ $R = pop(rstack)$ $gencode(n_2)$ $emit(R \leftarrow R\ op\ top(rstack))$ $push(R, rstack)$	label(right) < k label(left) ≥ label(right)
C4	$swap(rstack)$ $gencode(n_2)$ $R = pop(rstack)$ $gencode(n_1)$ $emit(top(rstack) \leftarrow top(rstack)\ op\ R)$ $push(R, rstack)$ $swap(rstack)$	label(left) < k label(right) > label(left)
C5	$gencode(n_2)$ $T = pop(tstack)$ $emit(T \leftarrow top(rstack))$ $gencode(n_1)$ $emit(top(rstack) \leftarrow top(rstack)\ op\ T)$ $push(T, tstack)$	

Code Generation with Two Registers r_0 and r_1



$r_1 \leftarrow c$

$r_0 \leftarrow d$

$r_0 \leftarrow r_0 - e$

$r_1 \leftarrow r_1 / r_0$

$g(/) : C3$

$g(+) : C4$

Control Stack

C1
```
emit(top(rstack) ← name)
```

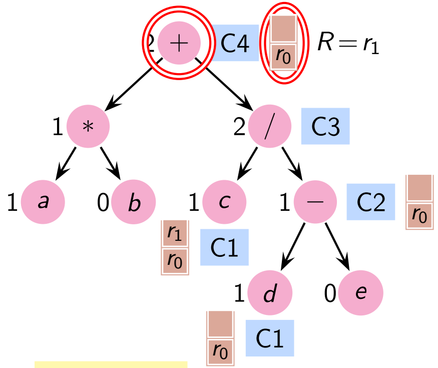
C2
```
gencode(n1)
emit(top(rstack) ← top(rstack) op name)
```

C3 — label(right) < k; label(left) ≥ label(right)
```
gencode(n1)
R = pop(rstack)
gencode(n2)
emit(R ← R op top(rstack))
push(R, rstack)
```

C4 — label(left) < k; label(right) > label(left)
```
swap(rstack)
gencode(n2)
R = pop(rstack)
gencode(n1)
emit(top(rstack) ← top(rstack) op R)
push(R, rstack)
swap(rstack)
```

C5
```
gencode(n2)
T = pop(tstack)
emit(T ← top(rstack))
gencode(n1)
emit(top(rstack) ← top(rstack) op T)
push(T, tstack)
```

Code Generation with Two Registers r_0 and r_1



$r_1 \leftarrow c$
$r_0 \leftarrow d$
$r_0 \leftarrow r_0 - e$
$r_1 \leftarrow r_1 / r_0$

$g(+) : C4$

Control Stack

C1	$emit(top(rstack) \leftarrow name)$	
C2	$gencode(n_1)$ $emit(top(rstack) \leftarrow top(rstack)\ op\ name)$	
C3	$gencode(n_1)$ $R = pop(rstack)$ $gencode(n_2)$ $emit(R \leftarrow R\ op\ top(rstack))$ $push(R, rstack)$	label(right) $<$ k label(left) $\geq$ label(right)
C4	$swap(rstack)$ $gencode(n_2)$ $R = pop(rstack)$ $gencode(n_1)$ $emit(top(rstack) \leftarrow top(rstack)\ op\ R)$ $push(R, rstack)$ $swap(rstack)$	label(left) $<$ k label(right) $>$ label(left)
C5	$gencode(n_2)$ $T = pop(tstack)$ $emit(T \leftarrow top(rstack))$ $gencode(n_1)$ $emit(top(rstack) \leftarrow top(rstack)\ op\ T)$ $push(T, tstack)$	

IIT Bombay
cs302: Implementation of Programming Languages

Topic:

Code Generation

Section:

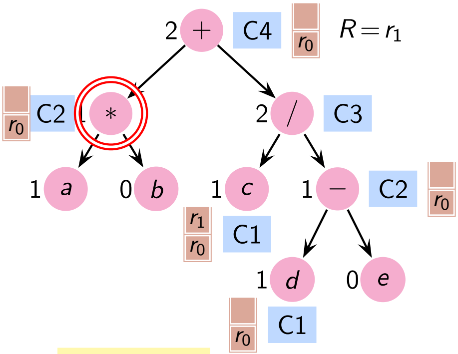
Global Register Allocation

Managing Registers Across Calls

Registers Usage in sclp

Instruction Selection

Code Generation with Two Registers r_0 and r_1



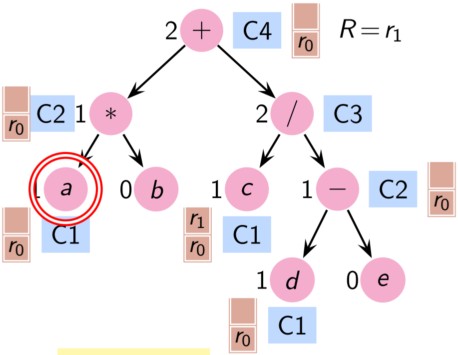
$r_1 \leftarrow c$
$r_0 \leftarrow d$
$r_0 \leftarrow r_0 - e$
$r_1 \leftarrow r_1 / r_0$

$g(*) : C2$
$g(+) : C4$

Control Stack

C1	$emit(top(rstack) \leftarrow name)$
C2	$gencode(n_1)$ $emit(top(rstack) \leftarrow top(rstack)\ op\ name)$
C3	$gencode(n_1)$ label(right) $<$ k $R = pop(rstack)$ label(left) $\geq$ label(right) $gencode(n_2)$ $emit(R \leftarrow R\ op\ top(rstack))$ $push(R, rstack)$
C4	$swap(rstack)$ label(left) $<$ k $gencode(n_2)$ label(right) $>$ label(left) $R = pop(rstack)$ $gencode(n_1)$ $emit(top(rstack) \leftarrow top(rstack)\ op\ R)$ $push(R, rstack)$ $swap(rstack)$
C5	$gencode(n_2)$ $T = pop(tstack)$ $emit(T \leftarrow top(rstack))$ $gencode(n_1)$ $emit(top(rstack) \leftarrow top(rstack)\ op\ T)$ $push(T, tstack)$

Code Generation with Two Registers r_0 and r_1



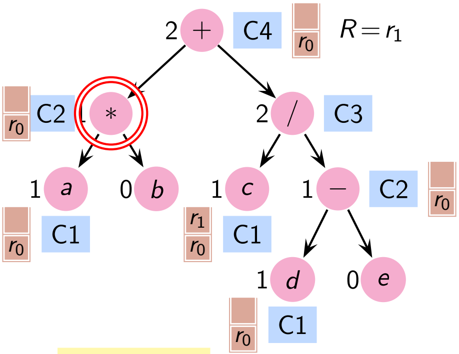
$r_1 \leftarrow c$
$r_0 \leftarrow d$
$r_0 \leftarrow r_0 - e$
$r_1 \leftarrow r_1 / r_0$
$r_0 \leftarrow a$

$g(a) : C1$
$g(*) : C2$
$g(+) : C4$

Control Stack

C1	$emit(top(rstack) \leftarrow name)$
C2	$gencode(n_1)$ $emit(top(rstack) \leftarrow top(rstack)\ op\ name)$
C3	$gencode(n_1)$ label(right) < k $R = pop(rstack)$ label(left) ≥ label(right) $gencode(n_2)$ $emit(R \leftarrow R\ op\ top(rstack))$ $push(R, rstack)$
C4	$swap(rstack)$ label(left) < k $gencode(n_2)$ label(right) > label(left) $R = pop(rstack)$ $gencode(n_1)$ $emit(top(rstack) \leftarrow top(rstack)\ op\ R)$ $push(R, rstack)$ $swap(rstack)$
C5	$gencode(n_2)$ $T = pop(tstack)$ $emit(T \leftarrow top(rstack))$ $gencode(n_1)$ $emit(top(rstack) \leftarrow top(rstack)\ op\ T)$ $push(T, tstack)$

Code Generation with Two Registers r_0 and r_1



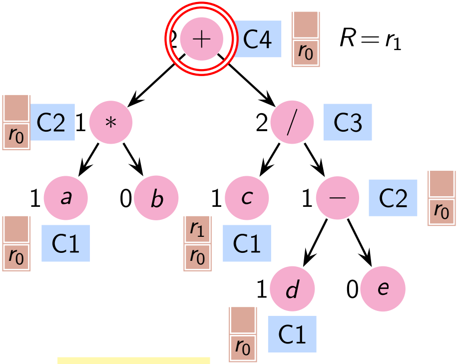
$r_1 \leftarrow c$
$r_0 \leftarrow d$
$r_0 \leftarrow r_0 - e$
$r_1 \leftarrow r_1 / r_0$
$r_0 \leftarrow a$
$r_0 \leftarrow r_0 * b$

$g(*) : C2$
$g(+) : C4$

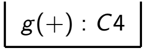
Control Stack

C1	emit(top(rstack) ← name)
C2	gencode(n_1) emit(top(rstack) ← top(rstack) op name)
C3	gencode(n_1) label(right) $<$ k R = pop(rstack) label(left) $\geq$ label(right) gencode(n_2) emit(R ← R op top(rstack)) push(R, rstack)
C4	swap(rstack) label(left) $<$ k gencode(n_2) label(right) $>$ label(left) R = pop(rstack) gencode(n_1) emit(top(rstack) ← top(rstack) op R) push(R, rstack) swap(rstack)
C5	gencode(n_2) T = pop(tstack) emit(T ← top(rstack)) gencode(n_1) emit(top(rstack) ← top(rstack) op T) push(T, tstack)

Code Generation with Two Registers r_0 and r_1



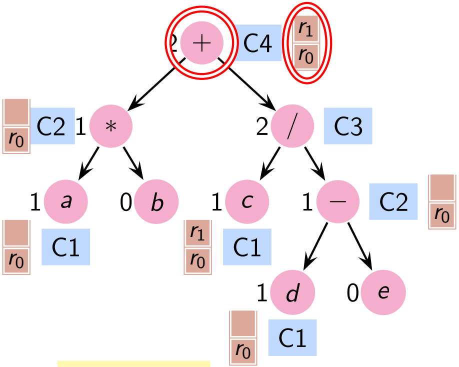
$r_1 \leftarrow c$
$r_0 \leftarrow d$
$r_0 \leftarrow r_0 - e$
$r_1 \leftarrow r_1 / r_0$
$r_0 \leftarrow a$
$r_0 \leftarrow r_0 * b$
$r_0 \leftarrow r_0 + r_1$



Control Stack

C1	$emit(top(rstack) \leftarrow name)$	
C2	$gencode(n_1)$ $emit(top(rstack) \leftarrow top(rstack)\ op\ name)$	
C3	$gencode(n_1)$ $R = pop(rstack)$ $gencode(n_2)$ $emit(R \leftarrow R\ op\ top(rstack))$ $push(R, rstack)$	label(right) $<$ k label(left) $\geq$ label(right)
C4	$swap(rstack)$ $gencode(n_2)$ $R = pop(rstack)$ $gencode(n_1)$ $emit(top(rstack) \leftarrow top(rstack)\ op\ R)$ $push(R, rstack)$ $swap(rstack)$	label(left) $<$ k label(right) $>$ label(left)
C5	$gencode(n_2)$ $T = pop(tstack)$ $emit(T \leftarrow top(rstack))$ $gencode(n_1)$ $emit(top(rstack) \leftarrow top(rstack)\ op\ T)$ $push(T, tstack)$	

Code Generation with Two Registers r_0 and r_1



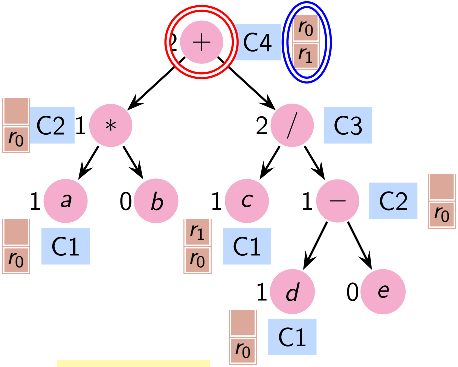
$r_1 \leftarrow c$
$r_0 \leftarrow d$
$r_0 \leftarrow r_0 - e$
$r_1 \leftarrow r_1 / r_0$
$r_0 \leftarrow a$
$r_0 \leftarrow r_0 * b$
$r_0 \leftarrow r_0 + r_1$

$g(+) : C4$

Control Stack

C1	$emit(top(rstack) \leftarrow name)$	
C2	$gencode(n_1)$ $emit(top(rstack) \leftarrow top(rstack)\ op\ name)$	
C3	$gencode(n_1)$ $R = pop(rstack)$ $gencode(n_2)$ $emit(R \leftarrow R\ op\ top(rstack))$ $push(R, rstack)$	label(right) $<$ k label(left) $\geq$ label(right)
C4	$swap(rstack)$ $gencode(n_2)$ $R = pop(rstack)$ $gencode(n_1)$ $emit(top(rstack) \leftarrow top(rstack)\ op\ R)$ $push(R, rstack)$ $swap(rstack)$	label(left) $<$ k label(right) $>$ label(left)
C5	$gencode(n_2)$ $T = pop(tstack)$ $emit(T \leftarrow top(rstack))$ $gencode(n_1)$ $emit(top(rstack) \leftarrow top(rstack)\ op\ T)$ $push(T, tstack)$	

Code Generation with Two Registers r_0 and r_1



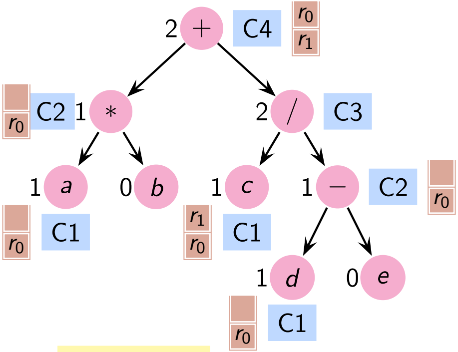
$r_1 \leftarrow c$
$r_0 \leftarrow d$
$r_0 \leftarrow r_0 - e$
$r_1 \leftarrow r_1 / r_0$
$r_0 \leftarrow a$
$r_0 \leftarrow r_0 * b$
$r_0 \leftarrow r_0 + r_1$

$g(+) : C4$

Control Stack

C1	$emit(top(rstack) \leftarrow name)$
C2	$gencode(n_1)$ $emit(top(rstack) \leftarrow top(rstack)\ op\ name)$
C3	$gencode(n_1)$ label(right) $<$ k $R = pop(rstack)$ label(left) $\geq$ label(right) $gencode(n_2)$ $emit(R \leftarrow R\ op\ top(rstack))$ $push(R, rstack)$
C4	$swap(rstack)$ label(left) $<$ k $gencode(n_2)$ label(right) $>$ label(left) $R = pop(rstack)$ $gencode(n_1)$ $emit(top(rstack) \leftarrow top(rstack)\ op\ R)$ $push(R, rstack)$ $swap(rstack)$
C5	$gencode(n_2)$ $T = pop(tstack)$ $emit(T \leftarrow top(rstack))$ $gencode(n_1)$ $emit(top(rstack) \leftarrow top(rstack)\ op\ T)$ $push(T, tstack)$

Code Generation with Two Registers r_0 and r_1



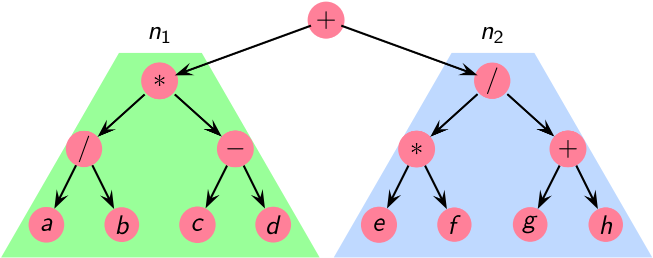
$r_1 \leftarrow c$
$r_0 \leftarrow d$
$r_0 \leftarrow r_0 - e$
$r_1 \leftarrow r_1 / r_0$
$r_0 \leftarrow a$
$r_0 \leftarrow r_0 * b$
$r_0 \leftarrow r_0 + r_1$

C1	$emit(top(rstack) \leftarrow name)$	
C2	$gencode(n_1)$ $emit(top(rstack) \leftarrow top(rstack)\ op\ name)$	
C3	$gencode(n_1)$ $R = pop(rstack)$ $gencode(n_2)$ $emit(R \leftarrow R\ op\ top(rstack))$ $push(R, rstack)$	label(right) $<$ k label(left) $\geq$ label(right)
C4	$swap(rstack)$ $gencode(n_2)$ $R = pop(rstack)$ $gencode(n_1)$ $emit(top(rstack) \leftarrow top(rstack)\ op\ R)$ $push(R, rstack)$ $swap(rstack)$	label(left) $<$ k label(right) $>$ label(left)
C5	$gencode(n_2)$ $T = pop(tstack)$ $emit(T \leftarrow top(rstack))$ $gencode(n_1)$ $emit(top(rstack) \leftarrow top(rstack)\ op\ T)$ $push(T, tstack)$	

Generate Code with Two Registers r_0 and r_1

Instructions

$r \leftarrow r \; op \; m$
$r_1 \leftarrow r_1 \; op \; r_2$
$r \leftarrow m$
$m \leftarrow r$



Generate Code with Two Registers r_0 and r_1

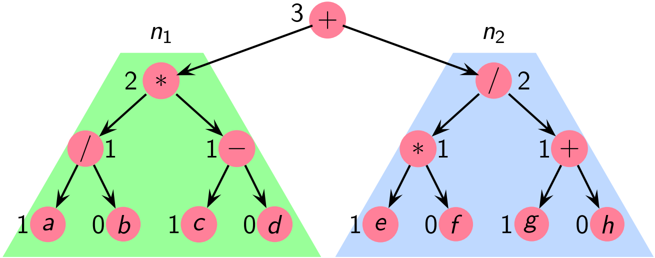
Instructions

$r \leftarrow r \ op \ m$

$r_1 \leftarrow r_1 \ op \ r_2$

$r \leftarrow m$

$m \leftarrow r$



Generate Code with Two Registers r_0 and r_1

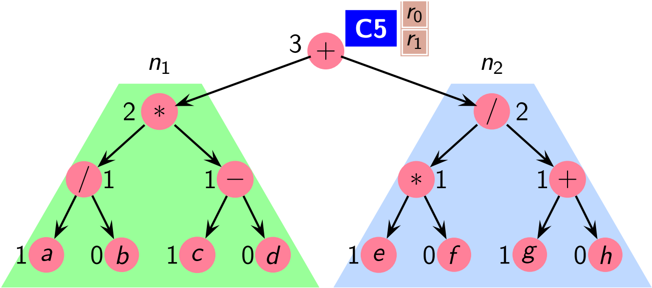
Instructions

$r \leftarrow r \; op \; m$

$r_1 \leftarrow r_1 \; op \; r_2$

$r \leftarrow m$

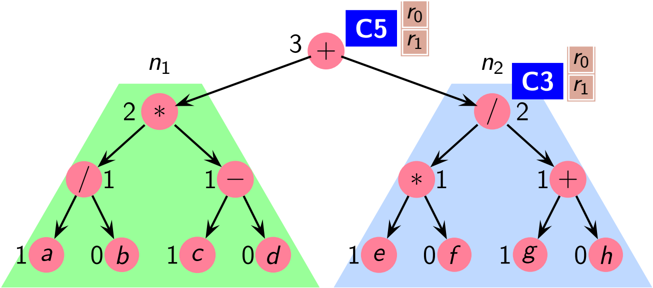
$m \leftarrow r$



Generate Code with Two Registers r_0 and r_1

Instructions

$r \leftarrow r\ op\ m$
$r_1 \leftarrow r_1\ op\ r_2$
$r \leftarrow m$
$m \leftarrow r$

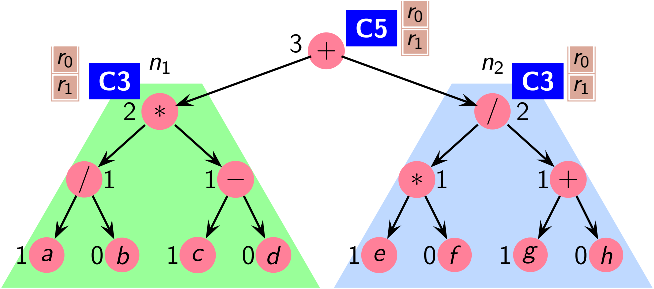


$r_0 \leftarrow e$
$r_0 \leftarrow r_0 * f$
$r_1 \leftarrow g$
$r_1 \leftarrow r_1 + h$
$r_0 \leftarrow r_0 / r_1$
$t_0 \leftarrow r_0$

Generate Code with Two Registers r_0 and r_1

Instructions

$r \leftarrow r\ op\ m$
$r_1 \leftarrow r_1\ op\ r_2$
$r \leftarrow m$
$m \leftarrow r$



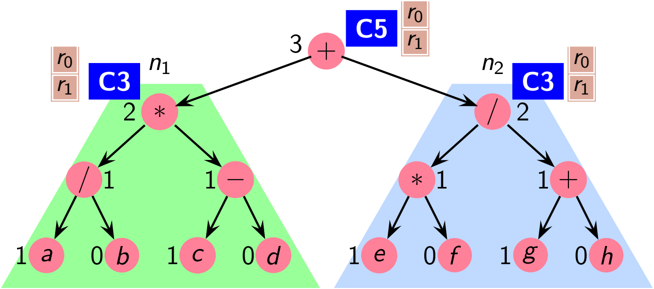
$r_0 \leftarrow e$
$r_0 \leftarrow r_0 * f$
$r_1 \leftarrow g$
$r_1 \leftarrow r_1 + h$
$r_0 \leftarrow r_0 / r_1$
$t_0 \leftarrow r_0$

$r_0 \leftarrow a$
$r_0 \leftarrow r_0 / b$
$r_1 \leftarrow c$
$r_1 \leftarrow r_1 - d$
$r_0 \leftarrow r_0 * r_1$

Generate Code with Two Registers r_0 and r_1

Instructions

$r \leftarrow r \ op \ m$
$r_1 \leftarrow r_1 \ op \ r_2$
$r \leftarrow m$
$m \leftarrow r$



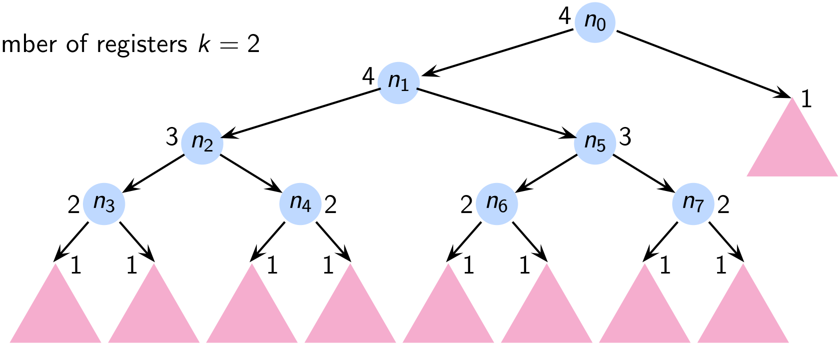
$r_0 \leftarrow e$
$r_0 \leftarrow r_0 * f$
$r_1 \leftarrow g$
$r_1 \leftarrow r_1 + h$
$r_0 \leftarrow r_0 / r_1$
$t_0 \leftarrow r_0$

$r_0 \leftarrow a$
$r_0 \leftarrow r_0 / b$
$r_1 \leftarrow c$
$r_1 \leftarrow r_1 - d$
$r_0 \leftarrow r_0 * r_1$

$r_0 \leftarrow r_0 + t_0$

Arguing Optimality: Dense Nodes and Major Nodes

Number of registers $k = 2$



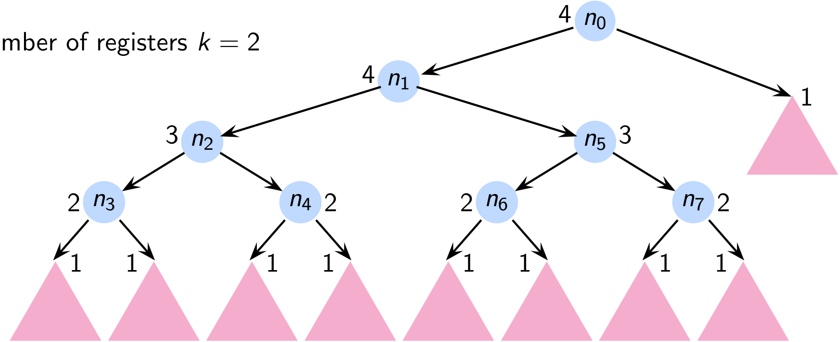
We define node n to be a

- *dense* node if label(n) $\geq k$
- *major* node if both its children are dense

  A major node falls in case 5 of the algorithm

Arguing Optimality: Dense Nodes and Major Nodes

Number of registers $k = 2$



We define node n to be a

- *dense* node if label(n) $\geq k$
- *major* node if both its children are dense

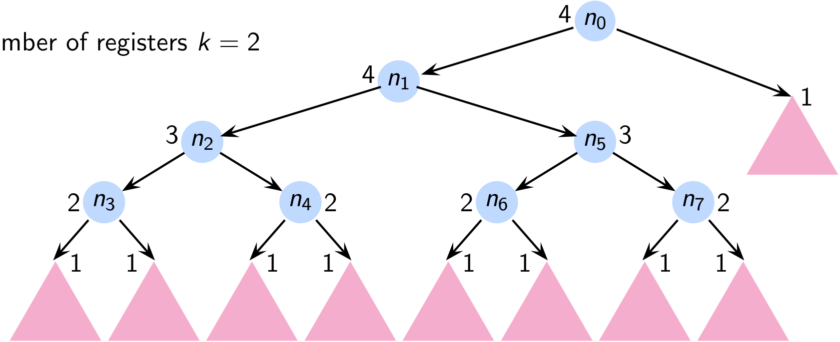
  A major node falls in case 5 of the algorithm

Dense Nodes $= \{n_0, n_1, n_2, n_3, n_4, n_5, n_6, n_7\}$

Major Nodes $= \{n_1, n_2, n_5\}$

Arguing Optimality: Dense Nodes and Major Nodes

Number of registers $k = 2$



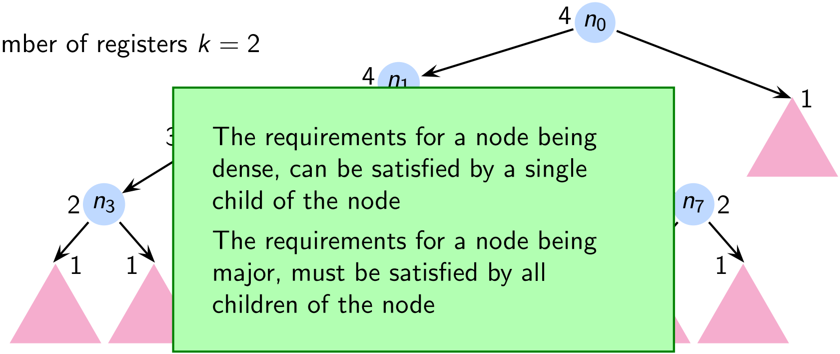
- Every major node is dense but not vice-versa
- The parent of every dense node is dense but the parent of every major node need not be major (e.g., n_1 is major but n_0 is not)

Dense Nodes $= \{n_0, n_1, n_2, n_3, n_4, n_5, n_6, n_7\}$

Major Nodes $= \{n_1, n_2, n_5\}$

Arguing Optimality: Dense Nodes and Major Nodes

Number of registers $k = 2$



The requirements for a node being dense, can be satisfied by a single child of the node

The requirements for a node being major, must be satisfied by all children of the node

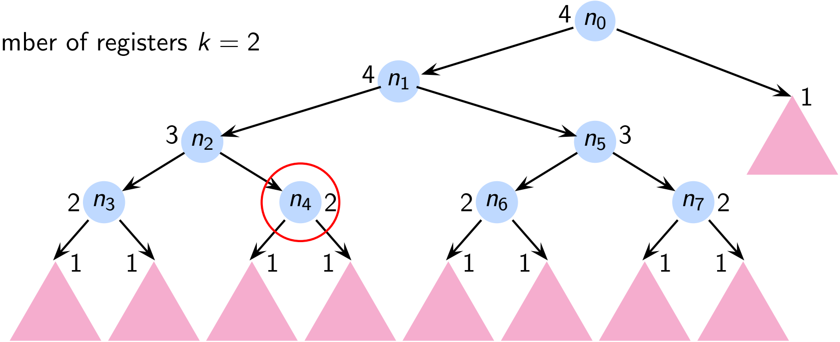
- Every major node is dense but not vice-versa
- The parent of every dense node is dense but the parent of every major node need not be major (e.g., n_1 is major but n_0 is not)

Dense Nodes $= \{n_0, n_1, n_2, n_3, n_4, n_5, n_6, n_7\}$

Major Nodes $= \{n_1, n_2, n_5\}$

Arguing Optimality: Dense Nodes and Major Nodes

Number of registers $k = 2$



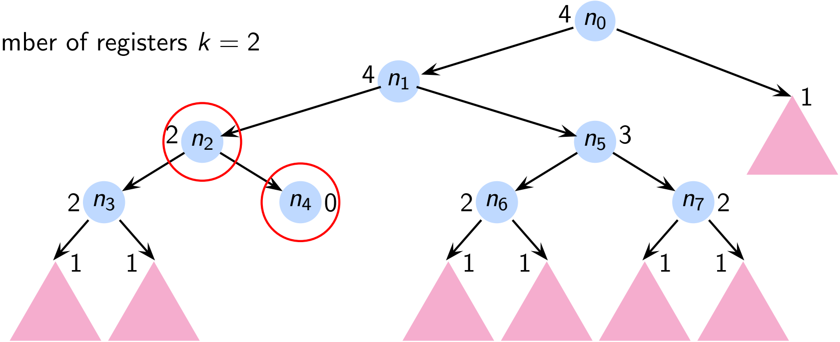
If we store n_4 in memory,

Dense Nodes $= \{n_0, n_1, n_2, n_3, n_4, n_5, n_6, n_7\}$

Major Nodes $= \{n_1, n_2, n_5\}$

Arguing Optimality: Dense Nodes and Major Nodes

Number of registers $k = 2$



If we store n_4 in memory,

- The $L(n_4)$ reduces to 0
- Hence, the label of n_2 reduces to 2. It remains dense but ceases to be major
- Node n_1 remains a major node

Dense Nodes $= \{n_0, n_1, n_2, n_3, n_5, n_6, n_7\}$

Major Nodes $= \{n_1, n_5\}$

Optimality of the Sethi-Ullman Algorithm

- The algorithm generates
  - exactly one instruction per operator node (i.e., every internal node)
  - exactly one load per left leaf
  - no load for any right leaf

Optimality of the Sethi-Ullman Algorithm

- The algorithm generates
  - exactly one instruction per operator node (i.e., every internal node)
  - exactly one load per left leaf
  - no load for any right leaf
- Thus, the optimality of the algorithm depends on not introducing extra stores

Optimality of the Sethi-Ullman Algorithm

- The algorithm generates
  - exactly one instruction per operator node (i.e., every internal node)
  - exactly one load per left leaf
  - no load for any right leaf
- Thus, the optimality of the algorithm depends on not introducing extra stores

  Consider an expression tree with m major nodes

Optimality of the Sethi-Ullman Algorithm

- The algorithm generates
    - exactly one instruction per operator node (i.e., every internal node)
    - exactly one load per left leaf
    - no load for any right leaf
- Thus, the optimality of the algorithm depends on not introducing extra stores

  Consider an expression tree with m major nodes
    - A store can reduce the number of major nodes by at most one

Optimality of the Sethi-Ullman Algorithm

- The algorithm generates
    - exactly one instruction per operator node (i.e., every internal node)
    - exactly one load per left leaf
    - no load for any right leaf
- Thus, the optimality of the algorithm depends on not introducing extra stores

  Consider an expression tree with m major nodes
    - A store can reduce the number of major nodes by at most one

      The node that becomes non-major, still remains a dense node so its parent remains a major node

Optimality of the Sethi-Ullman Algorithm

- The algorithm generates
    - exactly one instruction per operator node (i.e., every internal node)
    - exactly one load per left leaf
    - no load for any right leaf
- Thus, the optimality of the algorithm depends on not introducing extra stores

  Consider an expression tree with m major nodes
    - A store can reduce the number of major nodes by at most one
      The node that becomes non-major, still remains a dense node so its parent remains a major node
    - Hence the tree would need at least m stores regardless of the algorithm used for generating code

Optimality of the Sethi-Ullman Algorithm

- The algorithm generates
    - exactly one instruction per operator node (i.e., every internal node)
    - exactly one load per left leaf
    - no load for any right leaf
- Thus, the optimality of the algorithm depends on not introducing extra stores

  Consider an expression tree with m major nodes
    - A store can reduce the number of major nodes by at most one
      The node that becomes non-major, still remains a dense node so its parent remains a major node
    - Hence the tree would need at least m stores regardless of the algorithm used for generating code
    - The algorithm generates a single store for every major node (case 5), thus it generates exactly m stores

Optimality of the Sethi-Ullman Algorithm

- The algorithm generates
    - exactly one instruction per operator node (i.e., every internal node)
    - exactly one load per left leaf
    - no load for any right leaf
- Thus, the optimality of the algorithm depends on not introducing extra stores

  Consider an expression tree with m major nodes
    - A store can reduce the number of major nodes by at most one
      The node that becomes non-major, still remains a dense node so its parent remains a major node
    - Hence the tree would need at least m stores regardless of the algorithm used for generating code
    - The algorithm generates a single store for every major node (case 5), thus it generates exactly m stores
    - Since this is the smallest number of stores possible, the Sethi-Ullman algorithm is optimal