

Compiler (컴파일러)- Machine Code Generation & Machine Dependent Optimization

2015년 2학기
충남대학교 컴퓨터공학과
조은선

컴파일러 후반부 (Backend)

*Machine
-independent
Optimization*

Machine-independent
Optimization

*Virtual to physical
Mapping /
Machine-dependent
Optimization*

Instruction Selection

Instruction Scheduling

Register Allocation

Machine Code Emission/Opti

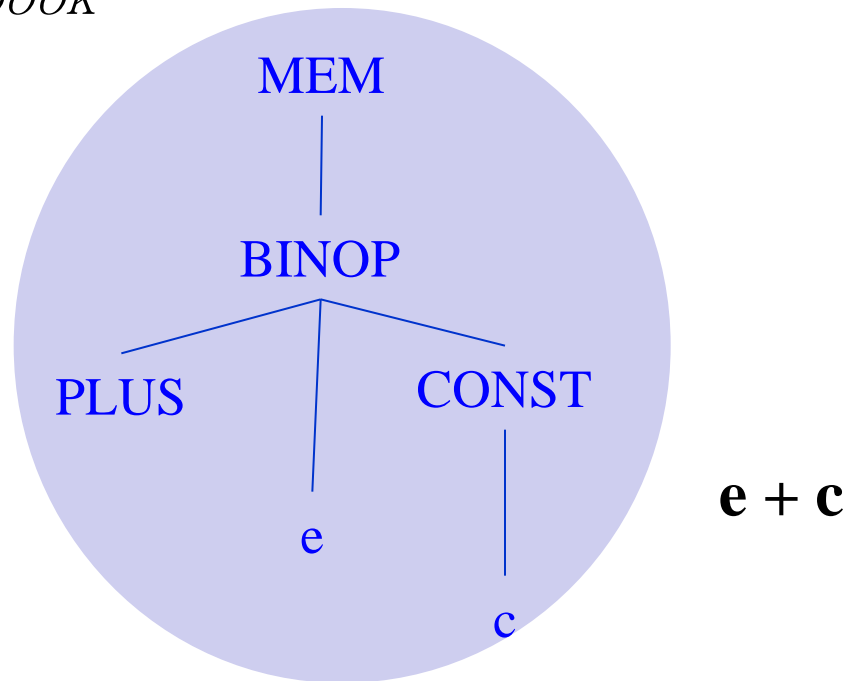
후반부 = 빠른 코드로 바꾸기 + 실제 머신에서 도는 코드로 바꾸기
(+ 코드 크기 줄이기)

Instruction Selection

Tree 기반 Intermediate Representation

- Low level한 경우도 있음.
 - With abstract machine instructions
 - Machine code 생성에 이용됨

예) *from Tiger book*



Tree 기반 Intermediate Representation

from Tiger book

MEM(e) : 주소 e 로 시작하는 메모리 한 word 의 내용. MOVE의 왼쪽에 사용되면 store, 다른 곳에 사용되면 fetch의 의미.

TEMP(t) : 레지스터 t

SEQ (s1, s2) : 문장 s1 수행후 s2 수행

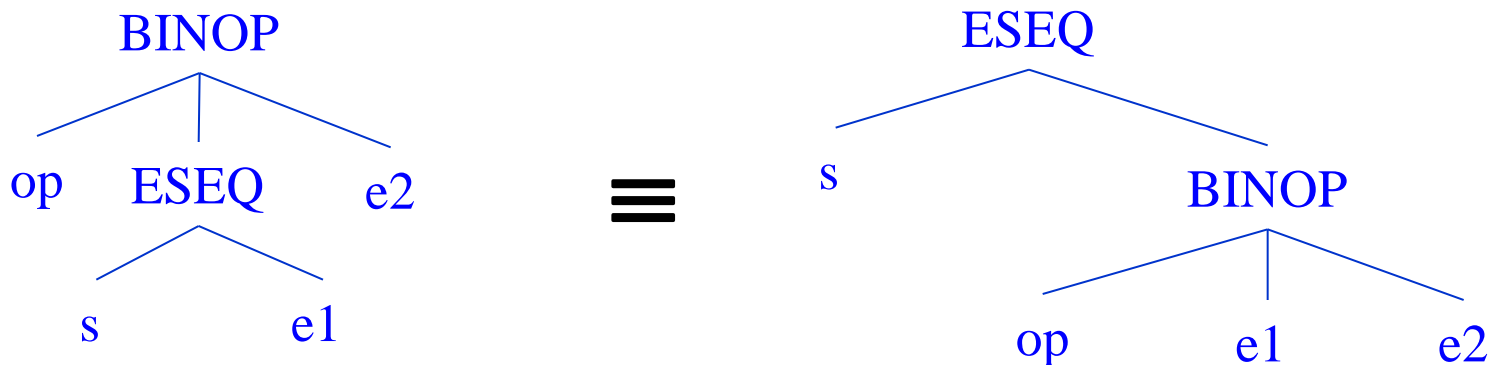
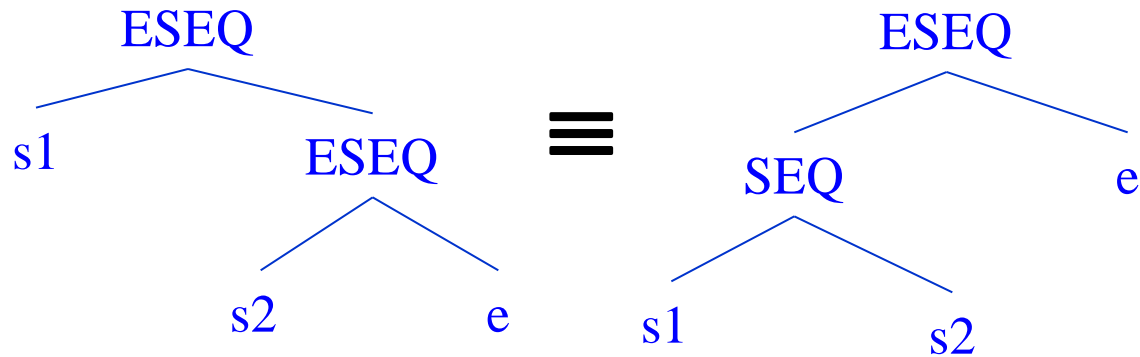
ESEQ(s,e) : 문장 s는 결과는 없고, side effect만 가지므로 ESEQ 전체의 결과를 내기 위해서는 e가 추가 수행됨.

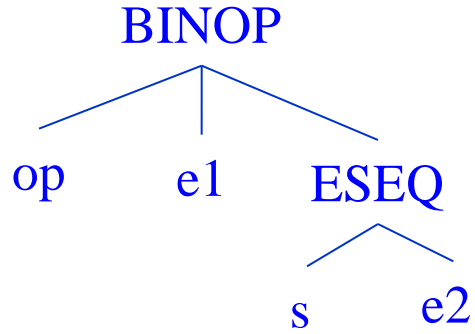
BINOP(o, e1, e2) : o는 PLUS, MINUS와 같은 binary operator. 결과는 피연산자 e1, e2를 가지고 이 o를 수행한 것. 결과는 메모리에 저장 후 주소 리턴

const(i) : 정수 상수 i

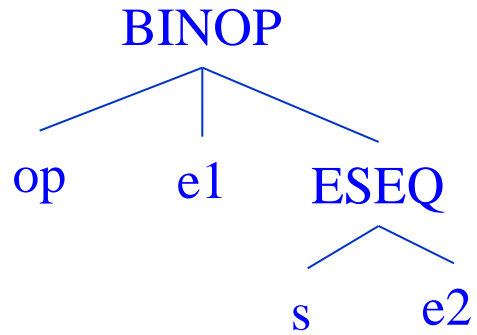
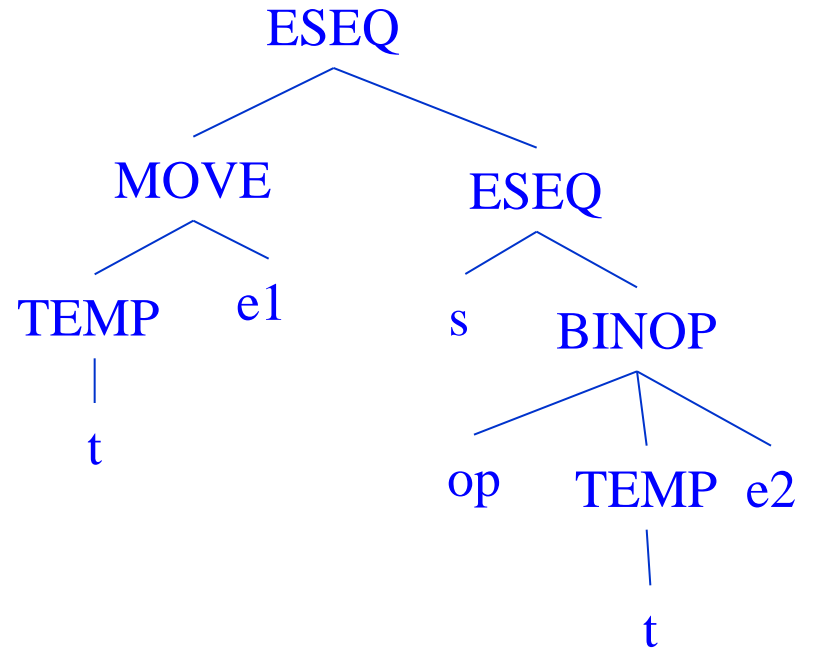
간단한 동치 관계

같은 의미를 가지는 것들 중 선택할 수 있다!

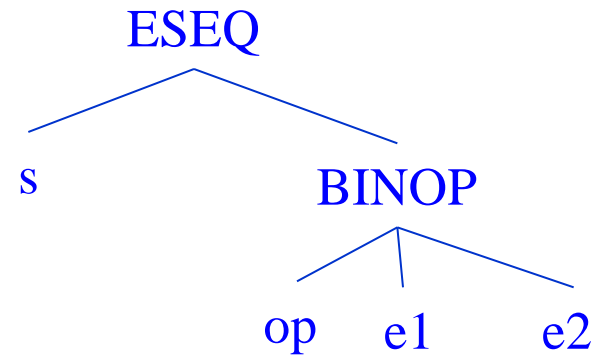




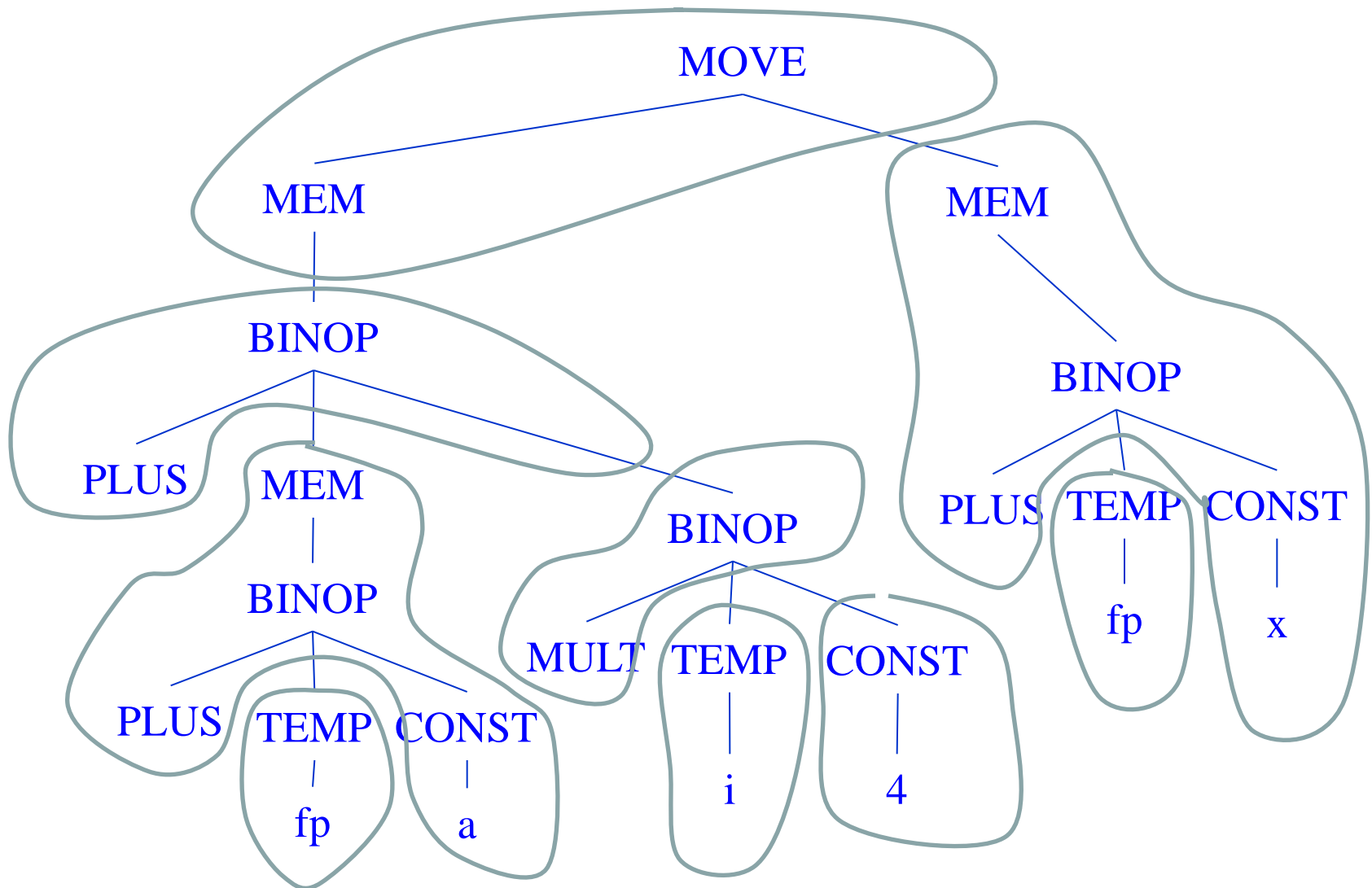
≡



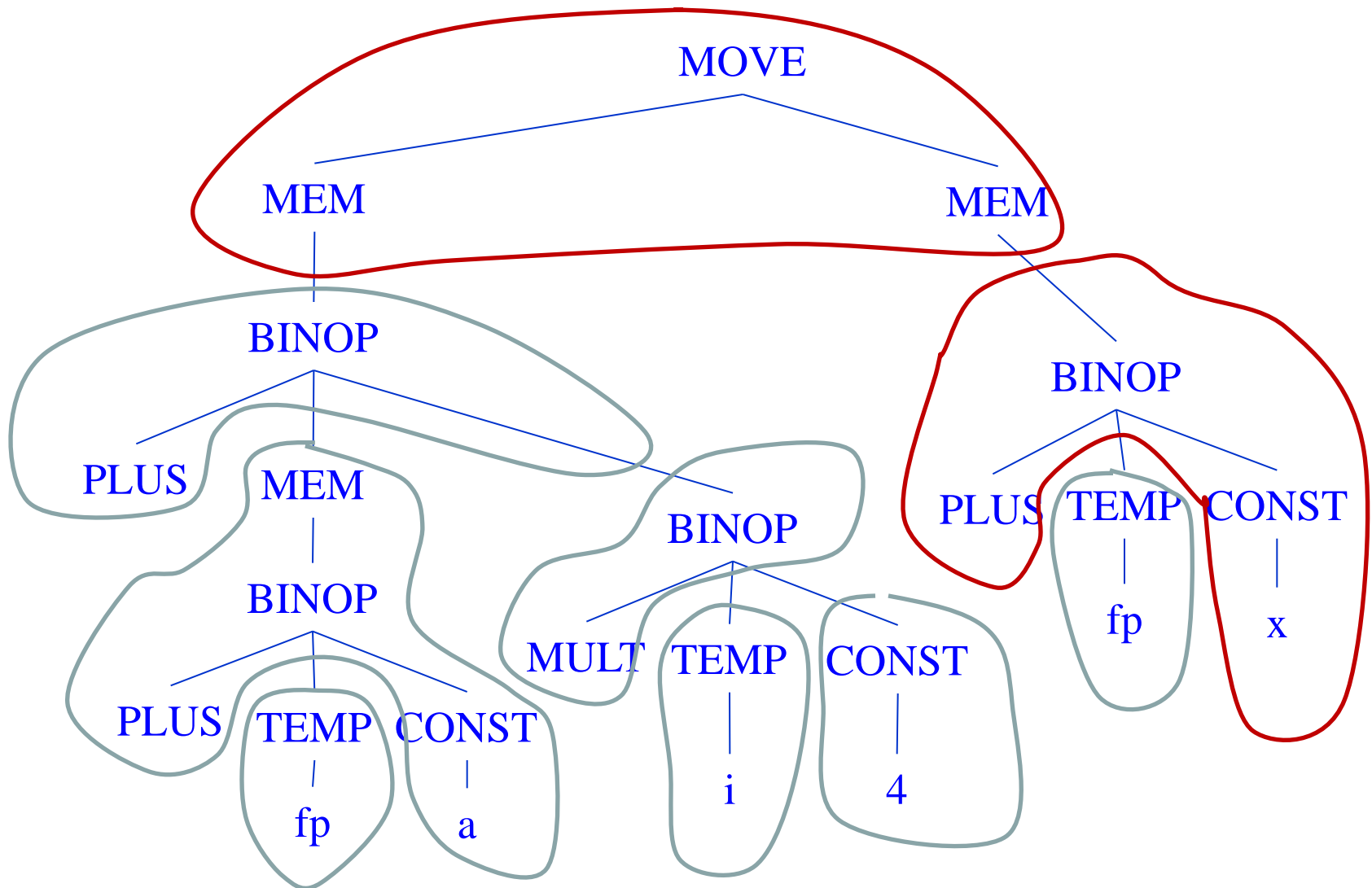
≡



More Instruction Selection (Option1)



More Instruction Selection (Option2)



Machine Code 형태로 보면

LOAD $r1 \leftarrow M[fp+a]$

ADDI $r2 \leftarrow r0 + 4$

MUL $r2 \leftarrow r1 \times r2$

ADD $r1 \leftarrow r1 + r2$

LOAD $r2 \leftarrow M[fp+x]$

STORE $M[r1+0] \leftarrow r2$

≡

LOAD $r1 \leftarrow M[fp+a]$

ADDI $r2 \leftarrow r0 + 4$

MUL $r2 \leftarrow r1 \times r2$

ADD $r1 \leftarrow r1 + r2$

LOAD $r2 \leftarrow fp + x$

STORE $M[r1] \leftarrow M[r2]$

Register Allocation

Low Level IR (복습)

- Machine independent code
- Opcode + operands
- Operands
 - Virtual registers- 무한히 있다고 가정
 - Special registers – stack pointer, pc, ...
 - Literals – 리터럴 상수 (크기 제약 없다고 가정)
 - Symbolic names – 주로 labels

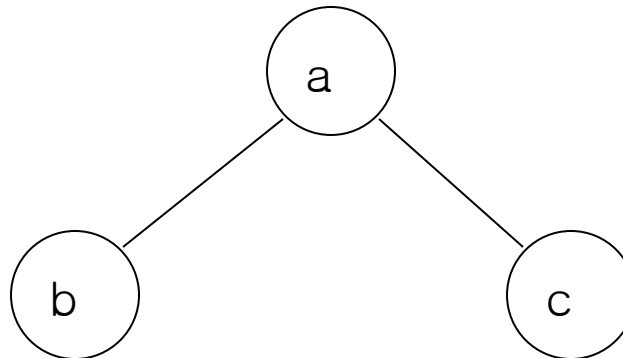
Register Allocation

- Motivation
 - Virtual register (VR) 의 개수는 무한하다고 가정했었음
 - 진짜 register의 개수는 유한 - machine 에 따라 다름
- Register allocation
 - VR을 최대한 physical register에 넣고 남는 것은 memory에
 - 처리속도 최대로 : 자주 사용되는 것을 physical register로
 - Spilling : virtual register 를 어쩔 수 없이 memory에 넣게 되는 것

Interference

- **Interference** : 서로 다른 두 definition이 live range 에서 공통 operation 을 가지고있는 경우
 - Live range : liveness analysis와 reaching definition analysis 등을 응용
- **Interference graph**
 - **Nodes** of the graph = variables
 - **Edges** : 서로 interfere 하면 연결

1: $a = 0$
2: $b = a$
3: $b * b$
4: $c = 2$
5: $a * c + 3$



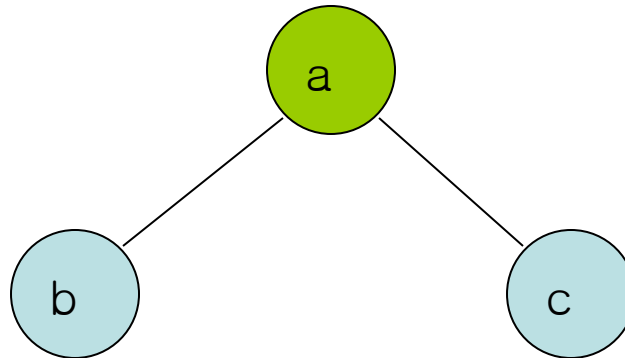
*For def1 $a = \{1,2,3,4,5\}$
For def2 $b = \{2,3\}$
For def4 $c = \{4,5\}$*



Graph Coloring

- Graph Coloring

- 변수 (virtual register)를 physical register에 할당 하는데 이용
- “Node에 색을 칠하는데, 연결된 node는 서로 다른 색으로 칠하기”
- 단순한 예
 - 레지스터 2개 : 2-coloring (색이 두개)

1: $a = 0$
2: $b = a$
3: $b * b$
4: $c = 2$
5: $a * c + 3$

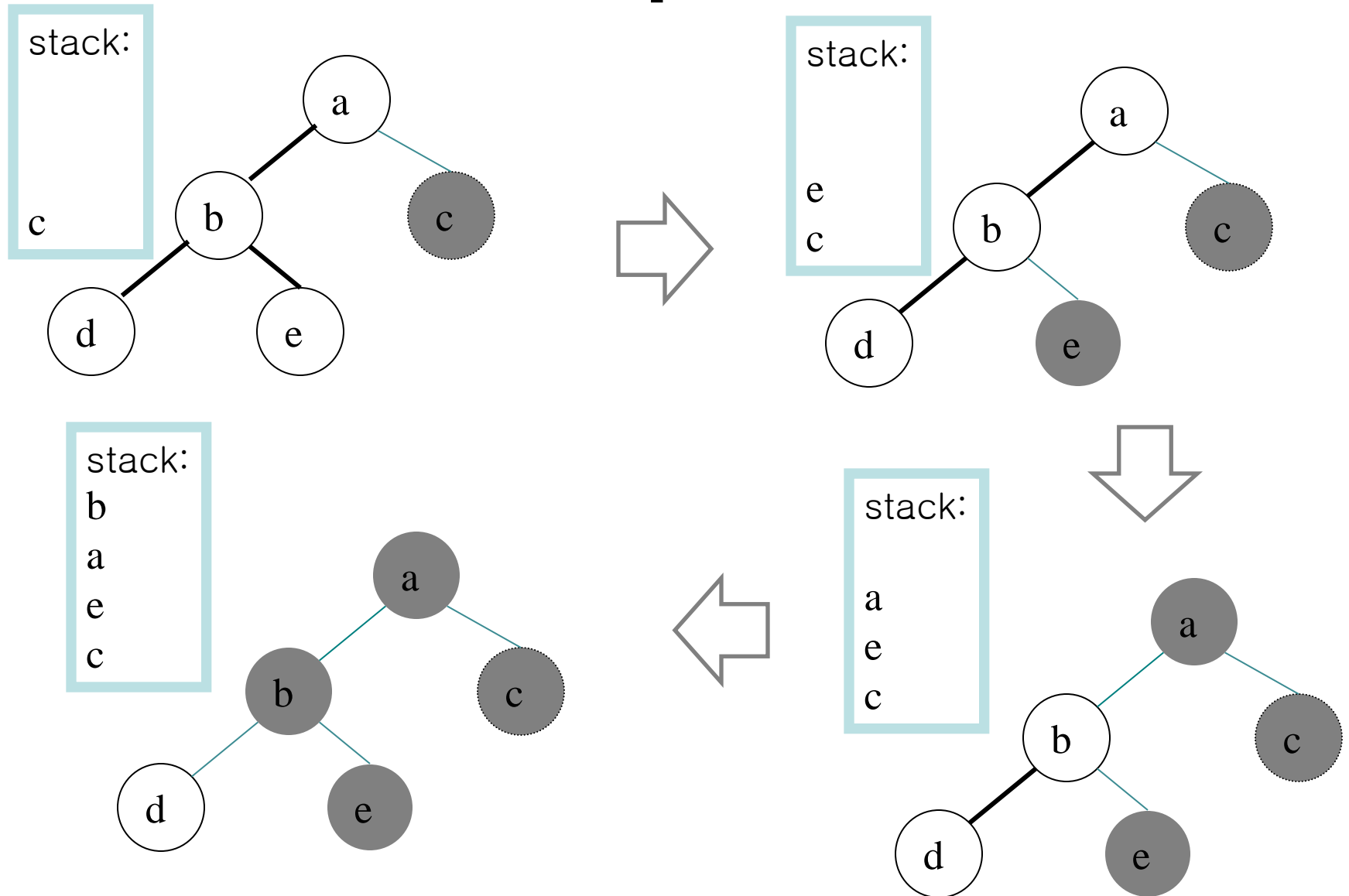


color	register
	eax
	ebx

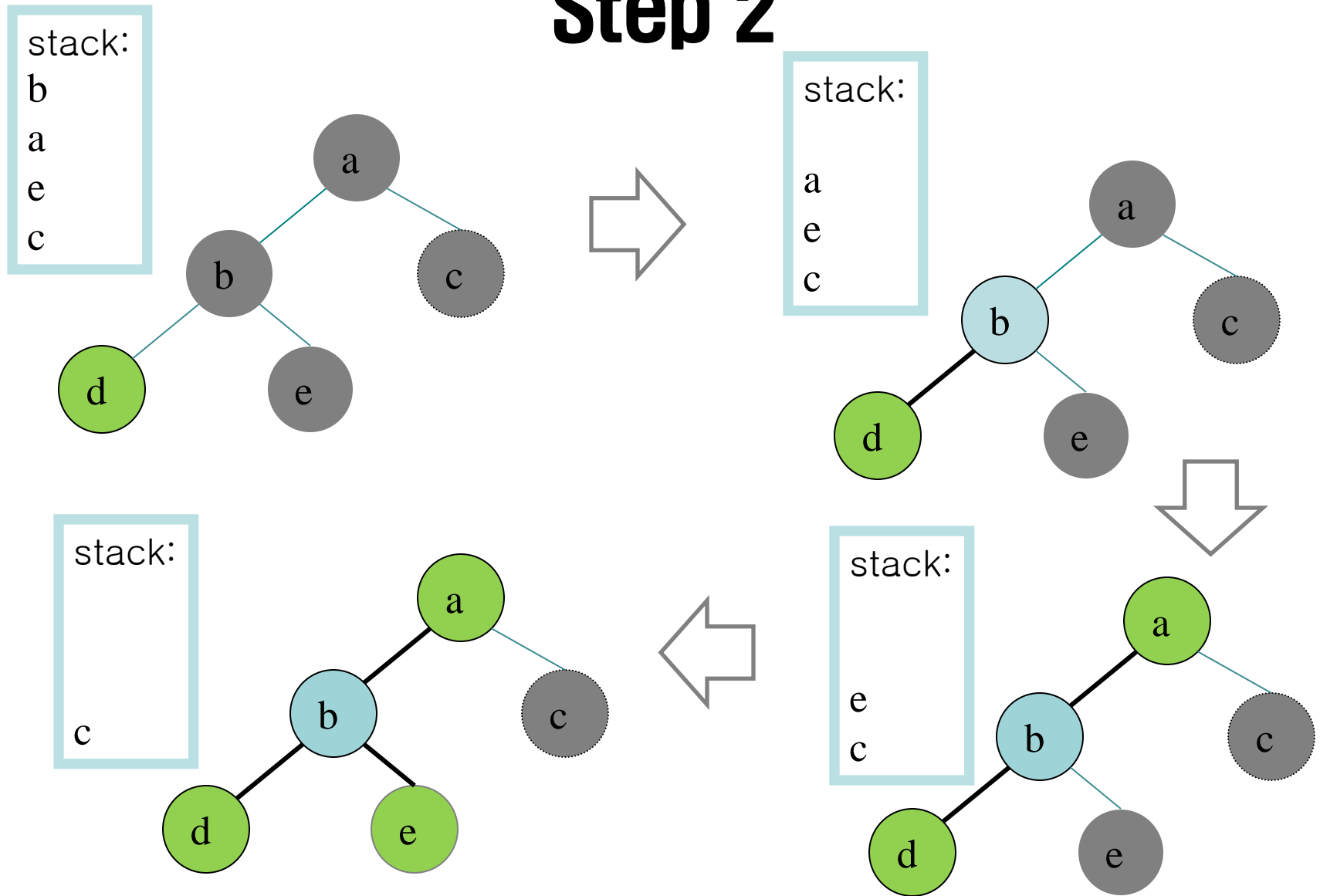
K-Graph Coloring Algorithm

- Kempe's algorithm [1879] --- 오래된 문제
- **Step 1 (simplify)** k 보다 작은 edge를 가진 node를 찾아서 edge와 함께 잘라냄
 - 이것을 stack에 저장
- **Step 2 (color)** 나머지 simplified subgraph로 쉽게 k-graph coloring을 할 수 있으면
 - stack에 있는 node를 pop해서 연결된 node에 없는 색을 칠하기를 반복
- **Step 3 (Spill) –optional** 안되면
 - Step1~step2 로 안 되는 경우도 많음
 - 사실 graph coloring은 NP-complete problem
 - 방법 : 변수 몇 개를 골라서 메모리에 저장

Step 1

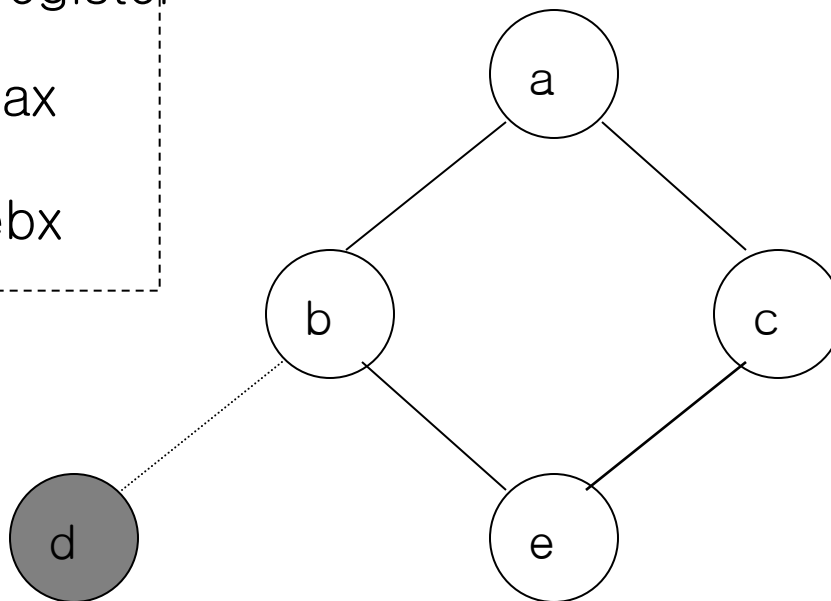
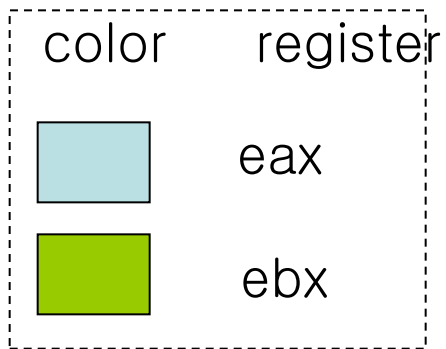


Step 2



Step 3 발생 경우 (1)

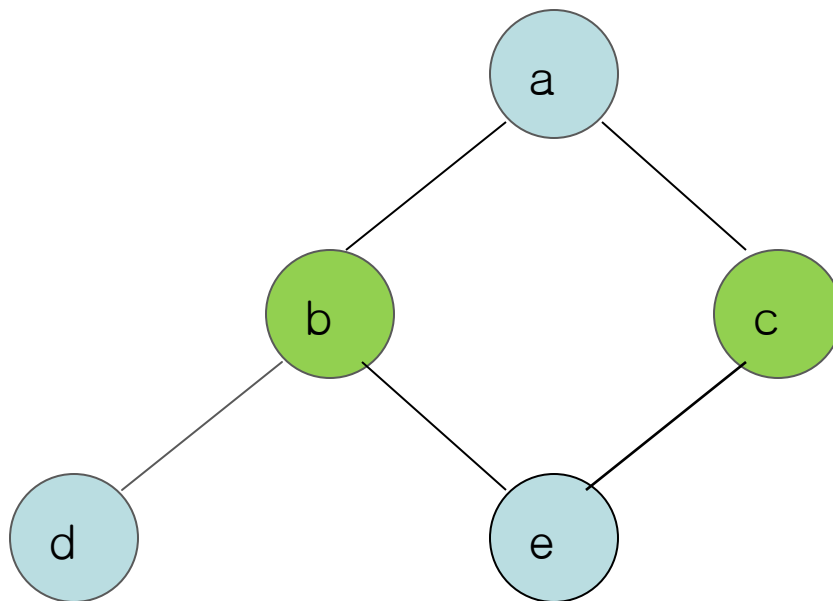
- Some lucky cases!



stack:
d

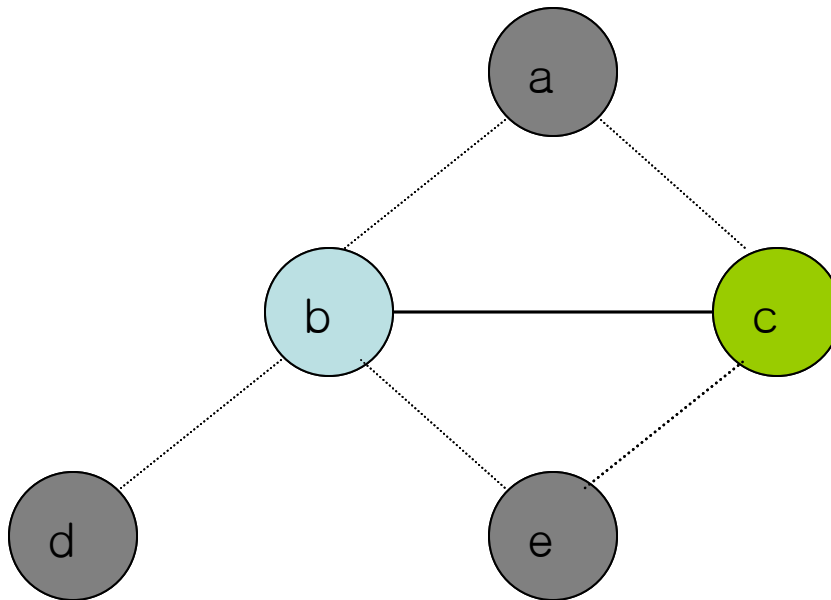
all nodes have
2 neighbours!

Lucky!



Step 3 발생 경우 (2)

- 그러나 k- color로 절대 coloring 할 수 없는 graph 도 물론 존재
➔ spilling!



no colors left for e or a !

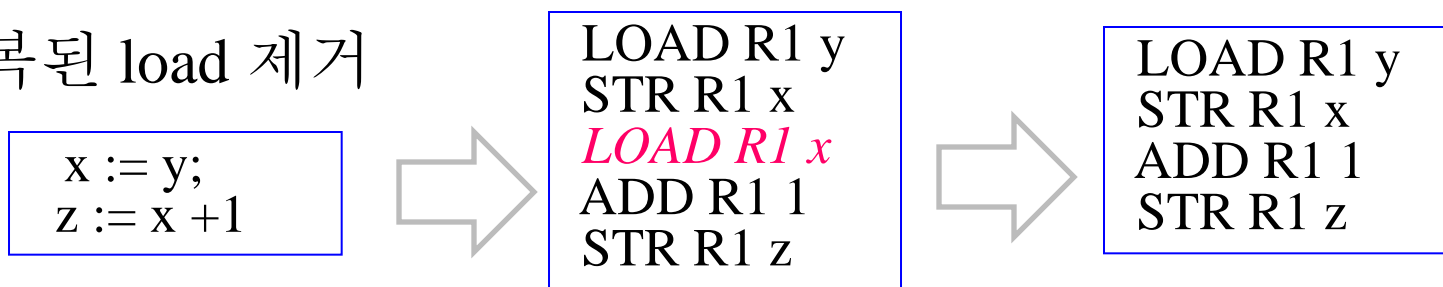
Spilling code

- Code rewriting
 - 새로운 temporary 변수 도입, 명령 재작성
- 예) t2를 spill 하기로 결정한 상태에서 ‘**add t1, t2**’ 하기
 - Spill을 결정한 t2 과 연결된 메모리 영역 정의
 - 예) 스택의 [ebp-24]
 - 새 임시 변수 t35를 도입
 - **mov t35, [ebp - 24]**
 - **add t1, t35**
 - note : t35의 live range는 매우 짧으므로 (한 두명령어) interference 가능성은 t2에 비해 거의 없다.

Other Machine Dependent Optimization

Machine Dependent Optimization

- 증감 연산이 있는 경우: `add r1 1` 대신 `inc r1`
- 특수 성질의 레지스터 활용 : `jump` 용 주소 레지스터
- 특수 목적의 명령어 활용 : 행렬 계산 명령어
 - Dual Core, Quad Core, Octo Core...
- Register 간 `mov` 제거
 - `Mov t1, t2` 가 있을 때, `t1`과 `t2`를 동일한 physical register로 allocate 됨
 - Interference 가능성이 증가되므로 병합 전 edge degree 합이 $k-1$ 이하 인지 확인
- 중복된 load 제거



“한학기 동안 수고 많았습니다”