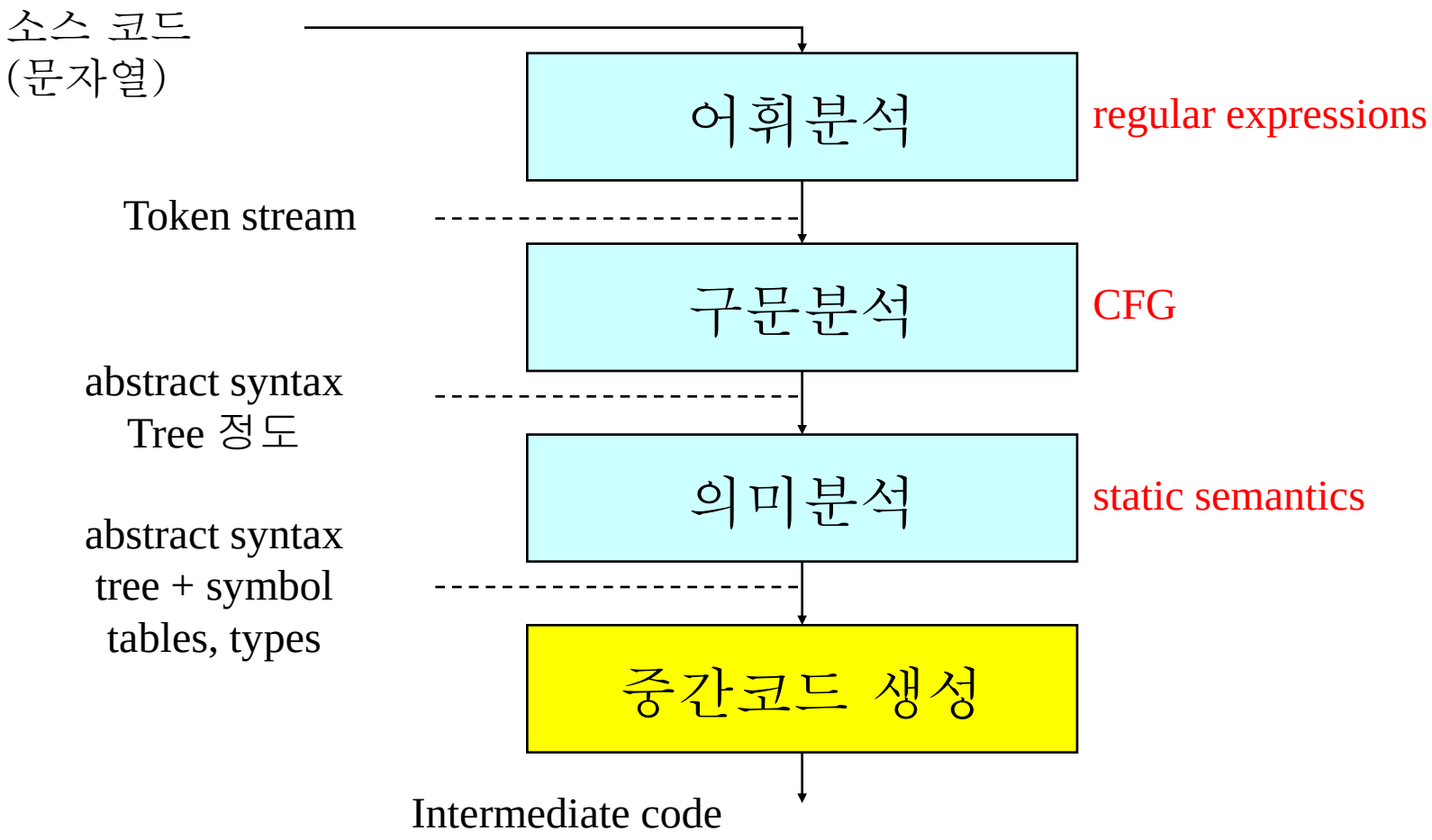
Compiler (컴파일러)
Intermediate Representation

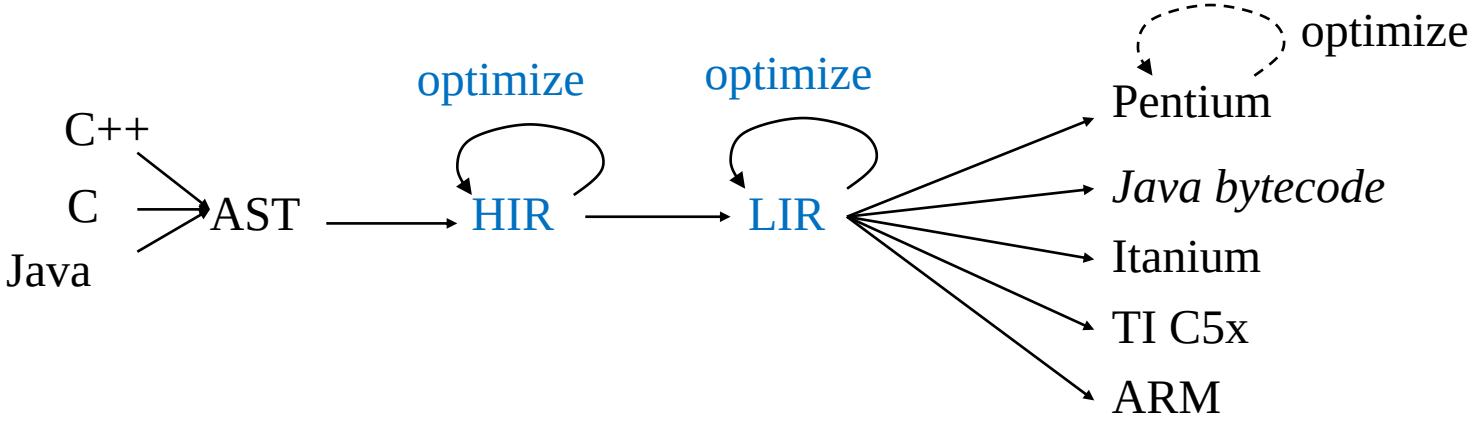
2015년 2학기
충남대학교 컴퓨터공학과
조은선

Where We Are...


소스 코드
(문자열)
어휘분석
regular expressions
Token stream
구문분석
CFG
abstract syntax
Tree 정도
의미분석
static semantics
abstract syntax
tree + symbol
tables, types
중간코드 생성
Intermediate code

Intermediate Representation (IR)

- Language-independent, machine-independent
  - 컴파일러 내부적으로 사용
- Tree나 Instruction list형태
  - Instruction (/ node)의 종류가 적어야 최적화/ 번역에 좋음
- 많은 경우 여러 개 사용 (multiple IR's)

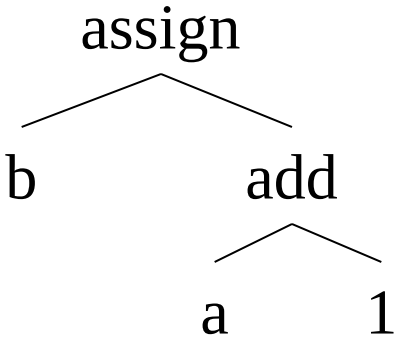


High Level IR

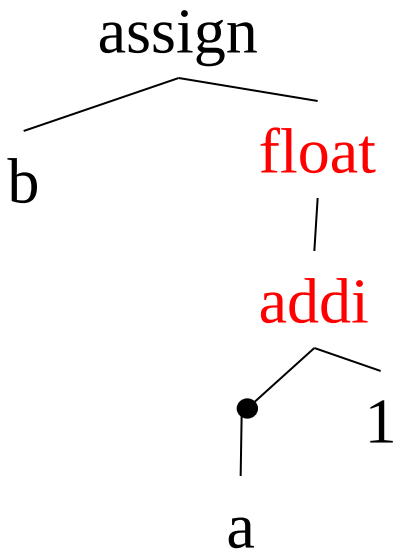
- High level과 Low level은 양분이 아닌, 상대적인 개념
  - 그리고, 3개 이상의 IR 도 사용가능함.
- High level IR : 여기서는 AST의 변형만 생각.
  - AST로부터의 번역이 용이
  - Input 언어의 모든 표현력을 그대로 유지
    - if, while, for, switch (structured control flow)
    - 변수, 표현식, 문장, 함수…
  - 따라서 input 언어에서 수행 가능한 언어 분석이 가능
    - 변수 사용 분석, loop transformation, 함수 inlinining

AST와 HIR의 비교

```
int a; float b;
...
b = a + 1;
```



(a) AST



(b) TCOL (HIR)

Low Level IR

- 주로 단순한 instruction으로 구성
  - Assembly어로 번역이 용이
- 가상기계 (주로 RISC) 를 emulate
- Machine 의 구조를 어느 정도 표현하게 됨
  - memory locations, registers, unstructured jump
- Instruction 예
  - Arithmetic/logic (a = b OP c), unary 연산, data movement (move, load, store), 함수 call/return, 분기 …

Low level IR의 종류

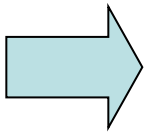
- N-tuple 표기법
  - (연산자, 피연산자1, 피연산자2) : triple
    - 기계어 비슷하나, instruction triple 자체를 결과로
  - (연산자, 피연산자1, 피연산자2, 결과) : quadruple
    - 즉, a = b OP c
    - 결과가 instruction과 따로 표현되므로 코드 이동이 용이해서 분석, optimize가 용이
    - 임시변수 다루어야
- Stack machine code
  - Java byte code, U-code : AST 로부터 생성이 용이
- Tree 표현
  - 기계어 생성 용이

- N-tuple (3-address) code √
- Stack machine code
- Tree code

N-Tuple, 3-주소 코드 (Quadruple)

- a = b OP c
  - 일반적으로 기계어가 가지는 피연산자 (또는 주소) 갯수 ≤ 3
- 예 : a = (b+ c) * (-e)

in high level language (or IR)



t1 = b + c

t2 = -e

a = t1 * t2

in 3-주소 코드

t1, t2는 컴파일러가 생성한 임시 변수

3-주소 코드 : Instructions

- Assignment instructions
  - a = b OP C (binary op)
    - arithmetic: ADD, SUB, MUL, DIV, MOD
    - logic: AND, OR, XOR
    - comparisons: EQ, NEQ, LT, GT, LEQ, GEQ
  - a = OP b (unary op)
    - arithmetic MINUS, logical NEG
  - a = b : copy instruction
  - a = [b] : load instruction
  - [a] = b : store instruction
  - a = addr b: symbolic address

3-주소 코드 : Instructions

- Flow of control
  - label L: label instruction
  - jump L: unconditional jump
  - cjump a L : conditional jump

- Function call
  - call f(a1, ..., an)
  - a = call f(a1, ..., an)

"결국, 일종의 가상 기계의 instruction"

3-주소 코드 : 피연산자

- 다음 중 하나
  - 프로그램 변수
  - 상수나 literals
  - 임시 변수
- 임시 변수 = 새 location
  - 중간 값 저장을 위해 사용
  - High level language 만큼 3-주소 코드가 expressive 하지 못하기 때문

Class Problem

다음 코드를

```
n = 0;
while (n < 10) {
        n = n+1;
}
n = n*2;
```

if, goto와 새로운 label을 써서 변환해보시오.
{} 와 같은 블럭구조는 사용하지 마시오.

예1) GCC의 GIMPLE

- GNU C 컴파일러의 표준 중간코드
  - 한가지 이상의 중간코드 존재 : GENERIC, GIMPLE, RTL ..

GCC : 이제는 Compiler Generation Framework

- `gcc/cc1` : Actual HLL-target specific driver/compiler!

예1) GCC GIMPLE

- GENERIC : 트리 형태 HIR

- GIMPLE : 3-주소 코드 (`gcc -fdump-tree-all test.c`)

  D.1954 = x * y , 또는

  gimple_assign <mult_exprt, D.1954, x, 10>

  - D.1954 는 임시변수

- RTL : 트리 형태의 LIR

GIMPLE Example : C-Like

(by `gcc -fdump-tree-all test.c`)

```c
#include <stdio.h>
void  main(){
        int x = 10;
        int y = 5;

        x = x*y + 3;
        printf("hello\n");
}
```

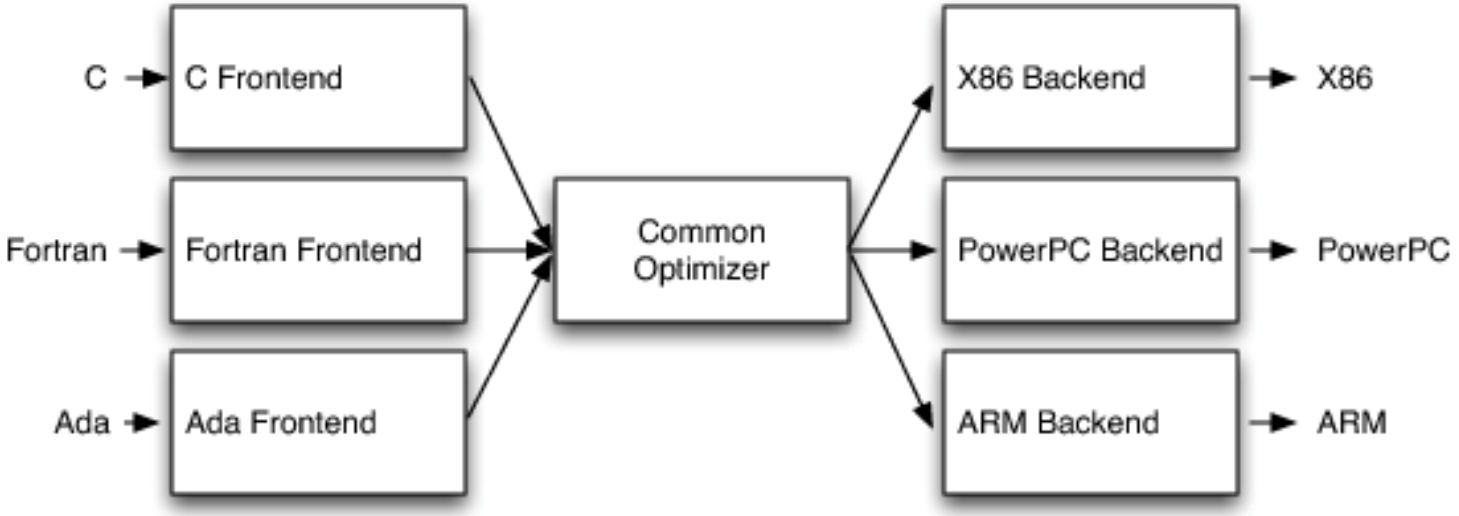
```
cse-03: ~/gcctest> ls
a.out              t.c.t06.vcg       t
t.c                t.c.t16.useless   t
t.c.t00.tu         t.c.t19.lower     t
t.c.t02.original   t.c.t20.eh
t.c.t03.gimple     t.c.t21.cfg
cse-03: ~/gcctest>
```

```
main ()
{
  int D.1779;
  int x;
  int y;

  x = 10;
  y = 5;
  D.1779 = x * y;
  x = D.1779 + 3;
  __builtin_puts (&"hello"[0]);
}
```

예 2) LLVM BIT 코드

- University of Illinois 에서 만든 컴파일러로 자체 IR을 보유
  - 현재는 Apple에서 지원
- LLVM IR : 언어와 machine에 독립적
  - 현재 gcc 기반의 C/C++ frontend (“Clang”)와 X86계열, ARM, SPARC 등의 backend 및 JIT compiler 포함



```
unsigned add1(unsigned a, unsigned b) {
  int c;
  return a+b;
}
```



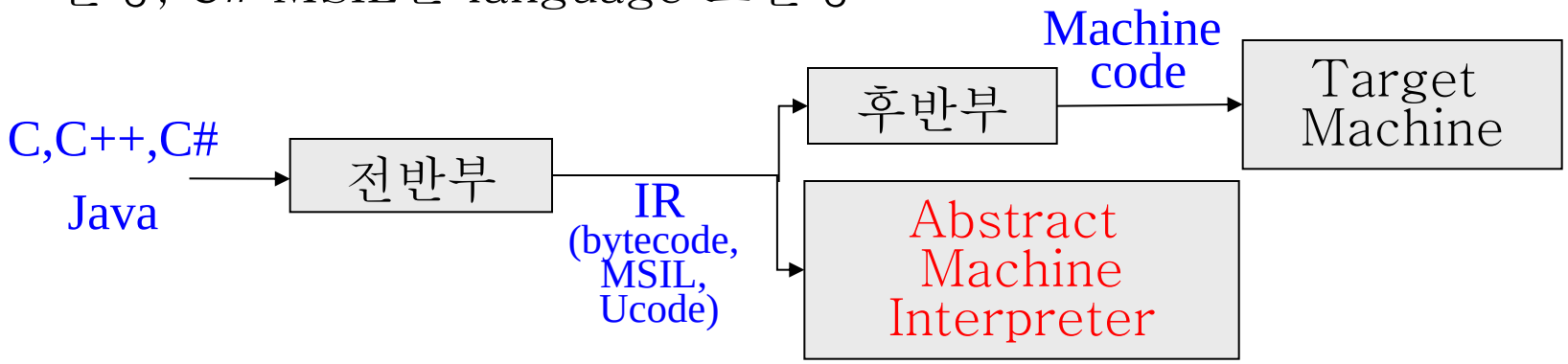
```
define i32 @add1(i32 %a, i32 %b) {
entry:
  %c = alloca i32, align 4
  %tmp1 = add i32 %a, %b
  ret i32 %tmp1
}
```

http://www.aosabook.org/en/llvm.html

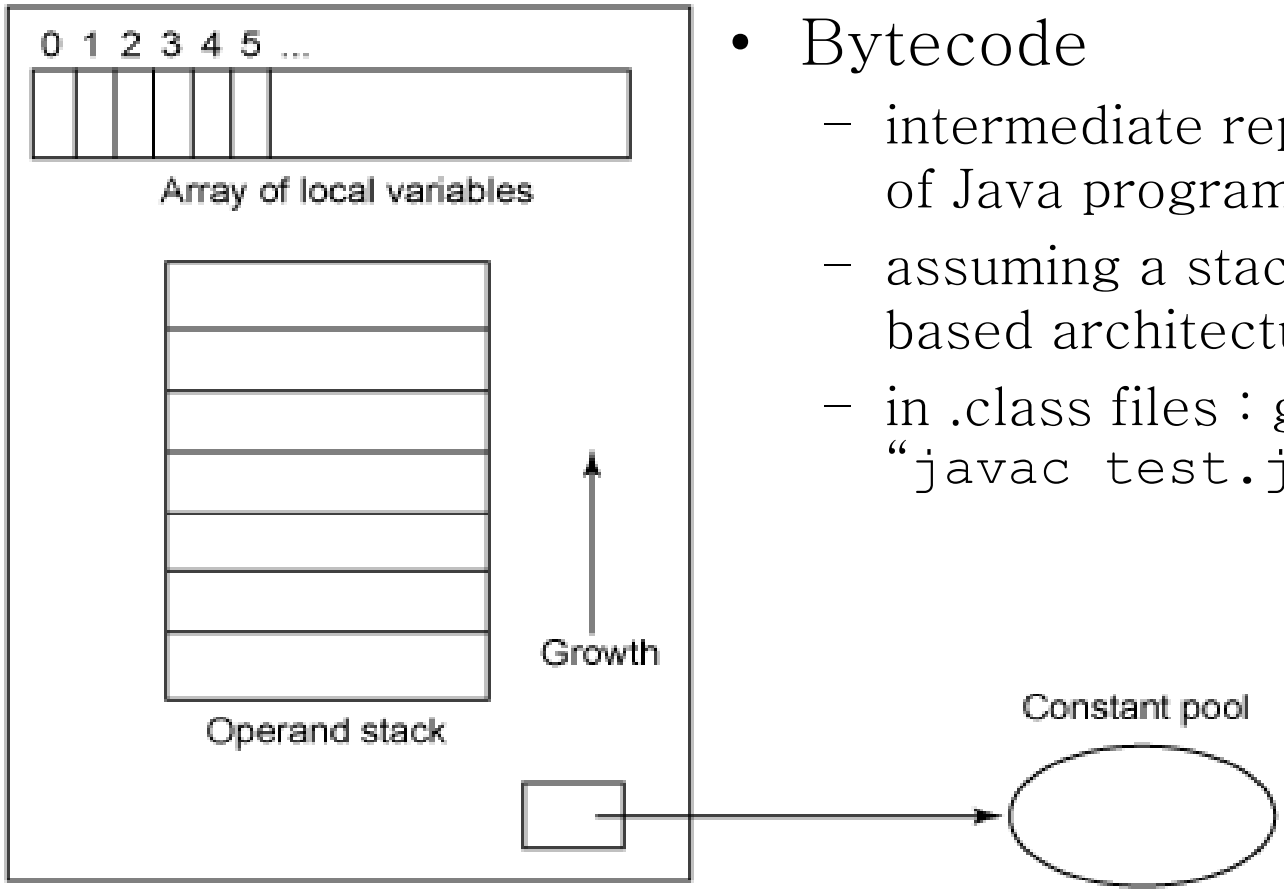
- N-tuple (3-address) code
- Stack machine code √
- Tree code

가상 기계 코드 (bytecode, MSIL, U-code)

- 컴파일러 전반부와 후반부를 나누는 잘 정의된 인터페이스를 정의하고,
  - 전반부 : high level language 에만 관계
  - 후반부 : machine code 에만 관계
- 가상기계에서 동작하도록 함.
  - stack machine 이 많음. (일반 IR로도 혼용되는 경우도 많음)
- 이식성, 호환성이 목적 : Java bytecode는 target machine 호환성, C# MSIL은 language 호환성



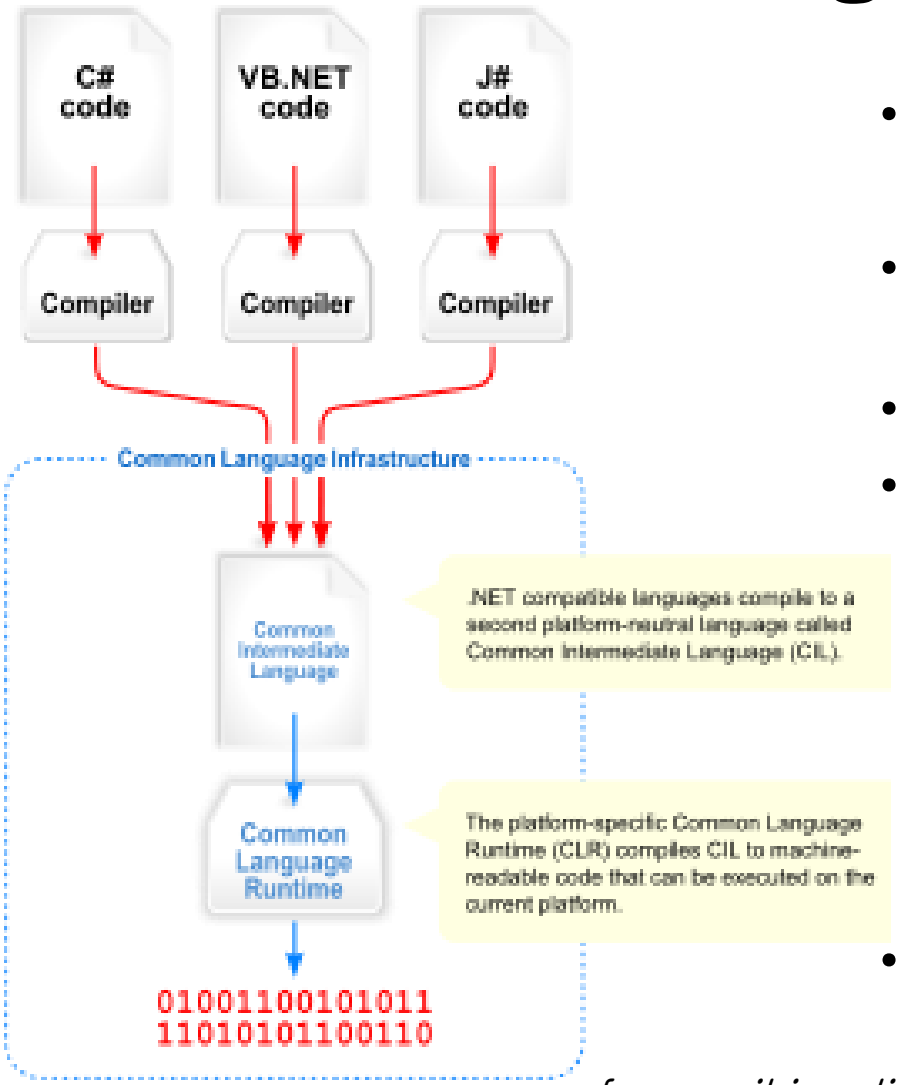
예1) JVM Bytecode



- Bytecode
  - intermediate representation of Java programs
  - assuming a stack machine based architecture
  - in .class files : generated by "`javac test.java`"

```
public Employee(String strName, int num)
{name = strName; idNumber = num; storeData(strName, num);}
Method Employee(java.lang.String,int)
0 aload_0 //push 'this' to stack
1 invokespecial #3 <Method java.lang.Object()> //call super
   class constructor
4 aload_0 //push 'this' to stack
5 aload_1 //push 'strName' to stack
6 putfield #5 <Field java.lang.String name> //push ref of
   'strName' to 'name' field of 'this'
9 aload_0 //push 'this' to stack
10 iload_2 //push 'num' to stack
11 putfield #4 <Field int idNumber> //push value of 'num' to
   'idNumber' field of 'this'
14 aload_0 //push 'this'
15 aload_1 //push ref of 'strName'
16 iload_2 //push ref of 'num'
17 invokespecial #6 <Method void
   storeData(java.lang.String, int)> //call a static method
20 return
```

예2) CIL (Common Intermediate Language)



from wikipedia

- MS .NET 프레임워크에서 사용됨 (예전 이름은 'MSIL')
- 여러 종류의 프로그래밍 언어가 공유하는 Low level IR
- C#에 맞추어 정의됨
- Common Language Infrastructure (CLI)
  - CTS (Common Type System)
  - Metadata
  - CLS (Common Language Specification)
  - VES (Virtual Execution System)
- VES 에서 CIL 을 기계어로 번역

예 2) CIL code (more)

```
add eax, edx                ; in x86
➔
        ldloc.0
        ldloc.1
        add
        stloc.0     // a = a + b or a += b;
```

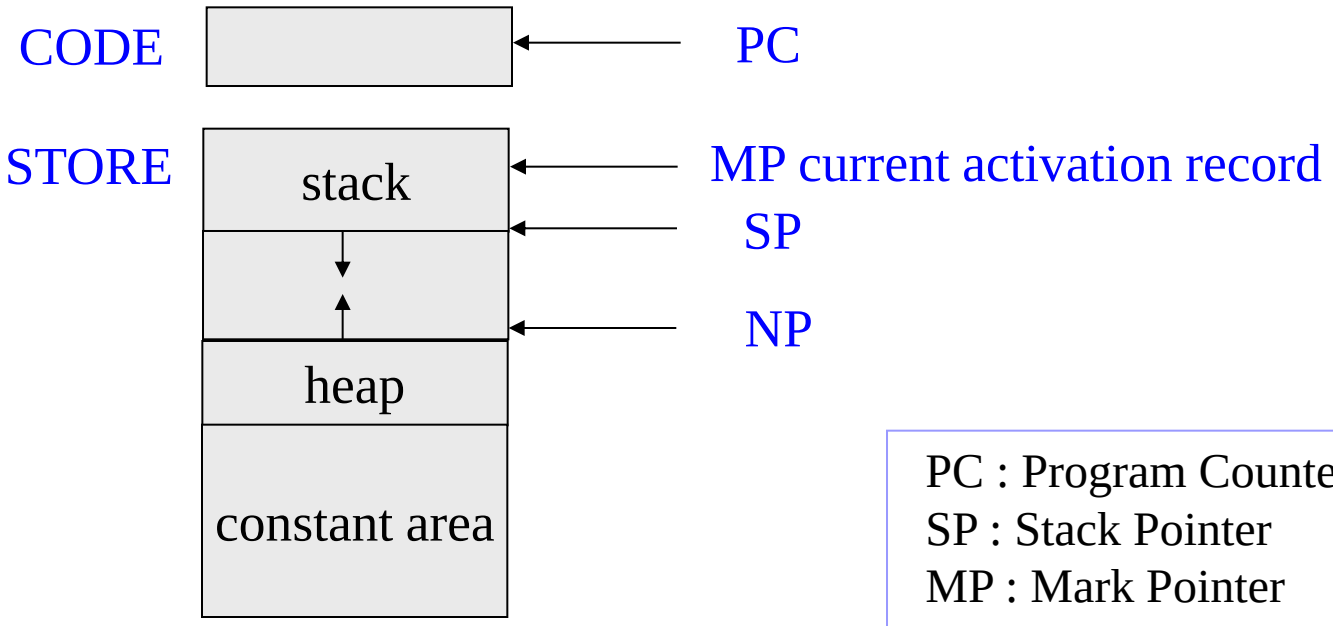
```
.assembly Hello {}
.assembly extern mscorlib {}
.method static void Main() {
        .entrypoint
        .maxstack 1
        ldstr "Hello, world!"
        call void [mscorlib]System.Console::WriteLine(string)
        ret
}
```

What will be the result?

from wikipedia

예3) 가상 기계 코드 : P-Code

- P-code
  - Pascal 컴파일러를 제작하는 데 사용된 중간언어
  - P-기계 = 특수레지스터 4개 + 기억공간



PC : Program Counter
SP : Stack Pointer
MP : Mark Pointer
NP : New Pointer

예 4) 가상기계코드-U-Code

- P-code의 변형
  - 교재에는 MiniC 를 위한 버전이 있음(!)
  - 인터프리터도 교재에 있음

- Example : x = a+ b*100

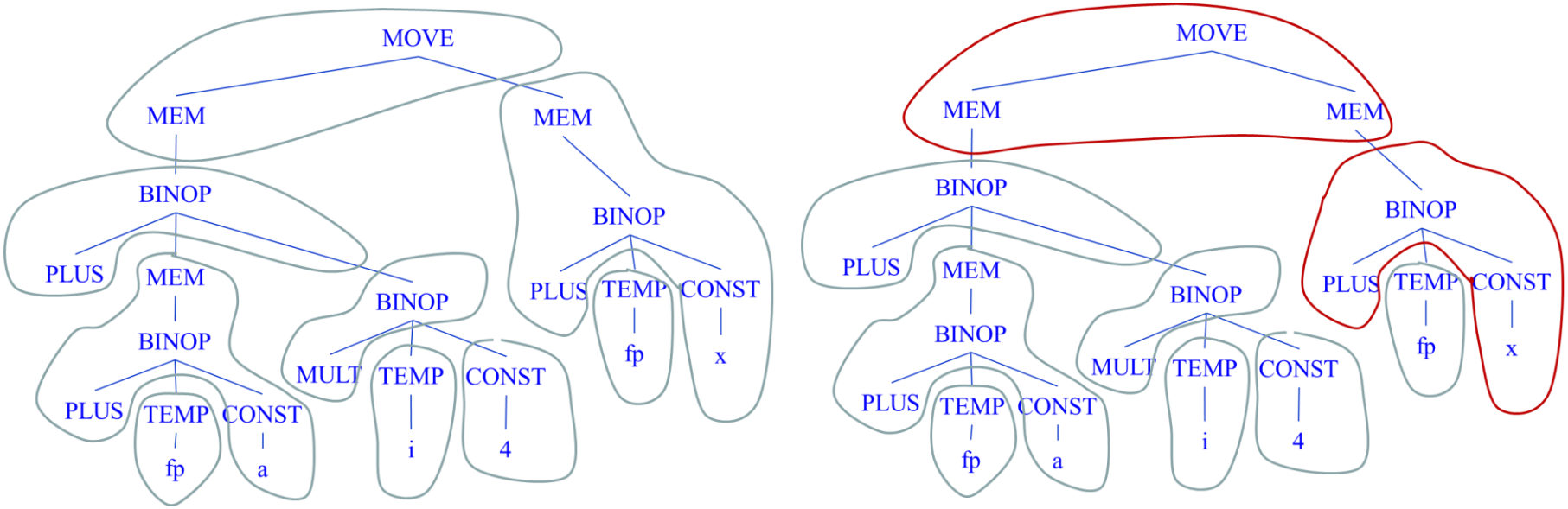
```
lod 1 1          /* a */
lod 1 2          /* b */
ldc 100
mult
add
str 1 3          /* x */
```

U-Code

- Instructions
  - unary – notop, neg, inc, dec, dup
  - binary – add, sub, mult, div, mod, swp,
    and, or, gt, lt, ge, le, eq, ne
  - stack-related – lod, str, ldc, lda
  - control flow – ujp, tjp, fjp
  - range check – chkh, chkl
  - indirect addressing – ldi, sti
  - function calls – call, ret, retv, ldp, proc, end
  - others – nop, bgn, sym

- N-tuple (3-address) code
- Stack machine code
- Tree code √

TREE 구조 코드



- AST나 HIR과 비슷한 면도 있으나 보다 자세함
  - 메모리 로드 등이 명시적으로 표현됨
- 기계어 코드 selection을 위해 사용

GCC 예) RTL Example : Tree structure (inside)

(by `gcc -fdump-tree-all-raw` test.c)

```
File  Edit  View  Search  Terminal  Help
     (nil))
(insn# 0 0 2 (set (reg:SI 0 ax [orig:59 D.1672 ] [59])
        (const_int 1 [0x1])) t.c:8# {*movsi_internal}
     (nil))
(insn# 0 0 2 (use (reg/i:SI 0 ax)) t.c:9#
     (nil))
(note# 0 0 NOTE_INSN_EPILOGUE_BEG)
(insn/f# 0 0 2 (parallel [
            (set (reg/f:SI 7 sp)
                (plus:SI (reg/f:SI 6 bp)
                    (const_int 4 [0x4])))
            (set (reg/f:SI 6 bp)
                (mem:SI (reg/f:SI 6 bp) [ S4 A8]))
            (clobber (mem:BLK (scratch) [ A8]))
        ]) t.c:9# {leave}
     (expr_list:REG_CFA_RESTORE (reg/f:SI 6 bp)
        (expr_list:REG_CFA_DEF_CFA (plus:SI (reg/f:SI 7 sp)
                (const_int 4 [0x4]))
            (nil))))
(jump_insn# 0 0 2 (return) t.c:9# {return_internal}
     (nil))
(barrier 0 0 0)
(note# 0 0 NOTE_INSN_DELETED)
```

- RTL (Register Transfer Language)
  - low level, machine dependent
  - representing program during compilation + specifying machine descriptions
  - 내부적으로 tree (rtx라고 부름) 운용, 접근 가능
  - 파일 .c.숫자.rtl 에 존재: generated by "`gcc test.c -da`"

    (mult : m (sign_extend : m x) (sign_extend : m y))
  - mult:m 의 m 등은 사용할 size

Translation from HIR to LIR

- to 3-주소 코드 (Quadruple) √
- to Stack machine code (U-code)- 과제
- to Tree 코드 instruction - *skip*

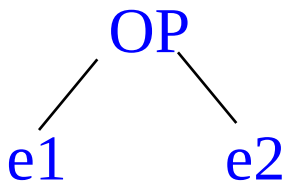
HIR to LIR

- Issues
  - HIR code 1줄 → LIR 코드 여러줄
    - eg. nested structures : nested while, if..
- Algorithmic way
  - HIR Tree 의 각 노드마다 LIR로 변환하는 방법 정의
- Notation
  - [[e]] : HIR e 에 대한 LIR expression
    - 주로 sequence of LIR instruction 임
  - t = [[e]] : e 가 표현식일 때, 결과값을 t에 저장
    - t = [[v]] : 변수 v 의 값을 (임시변수) t에 copy

3-주소 Translation 규칙
Arithmatic/Logic op

- Binary operations: t = [[e1 OP e2]]

t1 = [[e1]]
t2 = [[e2]]
t = t1 OP t2



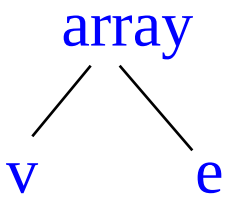
- Unary operations: t = [[OP e1]]

t1 = [[e1]]
t = OP t1

3-주소 Translation 규칙
Array Access

- Array access: t = [[v[e]]]
  - v : array[T] 타입
  - S : T 타입 데이터 하나의 크기

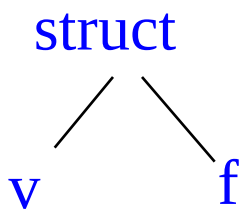
```
t1 = addr v
t2 = [[e]]
t3 = t2 * S
t4 = t1 + t3
t = [t4]    /* 주소로 load */
```



3-주소 Translation 규칙
Structure Access

- Structure access: t = [[v.f]]
  - v : T 타입
  - S : T 내에서 f 의 위치 (offset)

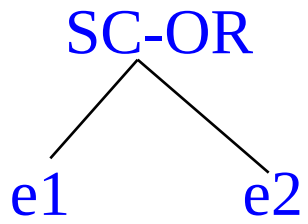
```
t1 = addr v
t2 = t1 + S
t = [t2]    /* 주소로 load */
```



3-주소 Translation 규칙
Short-cut OR

- Short-circuit OR: t = [[e1 SC-OR e2]]
  - e.g., || operator in C/C++

```
t = [[e1]]
cjump t Lend
t = [[e2]]
Lend:
```



1. evaluate e1
2. if e1 is true, then done
3. else evaluate e2

Class Problem

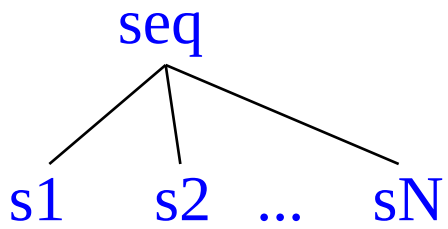
- Short-circuit AND: t = [[e1 SC-AND e2]]에 대한 translation 규칙을 구하라
  - e.g., && operator in C/C++

1. evaluate e1
2. if e1 is false, then done
3. else evaluate e2

3-주소 Translation 규칙
Statement Seq.

- Statement sequence: [[s1; s2; ...; sN]]

[[s1]]
[[s2]]
...
[[sN]]



단순 연결!

3-주소 Translation 규칙
Assignment

- Variable assignment: [[v = e]]

v = [[e]]

- Array assignment: [[v[e1] = e2]]

t1 = addr v
t2 = [[e1]]
t3 = t2 * S
t4 = t1 + t3
t5 = [[e2]]
[t4] = t5 /* 주소가 가리키는 곳에 store */

S = sizeof(T)
v : array(T)

3-주소 Translation 규칙
If

- [[if (e) then s]]

```
t1 = [[ e ]]
t2 = neg t1
cjump t2 Lend
[[ s ]]
Lend:
```

- [[if (e) then s1
  else s2]]

```
t1 = [[ e ]]
t2 = neg t1
cjump t2 Lelse
Lthen: [[ s1 ]]
jump Lend
Lelse: [[ s2 ]]
Lend:
```

3-주소 Translation 규칙
While

- [[while (e) s]]

while-do translation

Lloop: t1 = [[e]]
t2 = neg t1
cjump t2 Lend
[[s]]
jump Lloop
Lend:

3-주소 Translation 규칙
Switch

- [[switch (e) case v1:s1, ..., case vN:sN]]

```
t = [[ e ]]
L1: c =  t neq v1
cjump c L2
[[ s1 ]]
jump Lend   /* if there is a break */
L2: c =  t neq  v2
cjump c L3
[[ s2 ]]
jump Lend    /* if there is a break */
...
Lend:
```

Note: 테이블 lookup 으로 구현할 수도 있음
- 테이블은 target labels, 즉 L1, L2, L3을 가짐
-'t' 는 테이블 내의 한 entry를 뽑는 index.
-장점 : k 개분기가 1개로 줄어듬

3-주소 Translation 규칙
Function Call/Return

- [[call f(e1, e2, ..., eN)]]

  t1 = [[e1]]
  t2 = [[e2]]
  ...
  tN = [[eN]]
  call f(t1, t2, ..., tN)

- [[return e]]

  t = [[e]]
  return t

Statement Expressions

- Statement 도 expression 처럼 값을 가지도록 확장해보자
  - Block statements
  - If-then-else
  - Assignment statements
- Notation
  - t = [[s]] :LIR code for statement s
  - a sequence of LIR instructions
  - 임시변수 t 에 해당 s 의 결과값이 저장됨

3-주소 Translation 규칙
Statement Expression

- t = [[if (e) then s1 else s2]]

  t1 = [[e]]
  cjump t1 Lthen
  t = [[s2]]
  jump Lend
  Lthen: **t** = [[s1]]
  Lend:

- t = [[s1; s2; .. sN]]

  [[s1]]
  [[s2]]
  ...
  t = [[sN]]

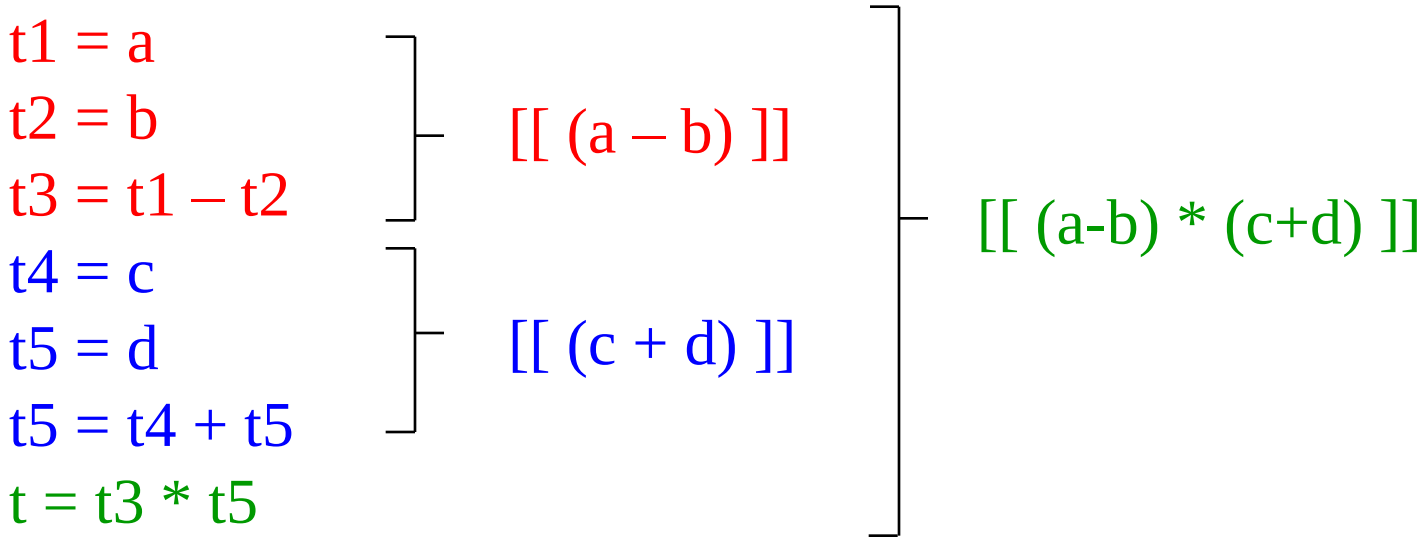
  - *결과값을 마지막 값으로*

- t = [[v = e]]

  v = [[e]]
  t = v

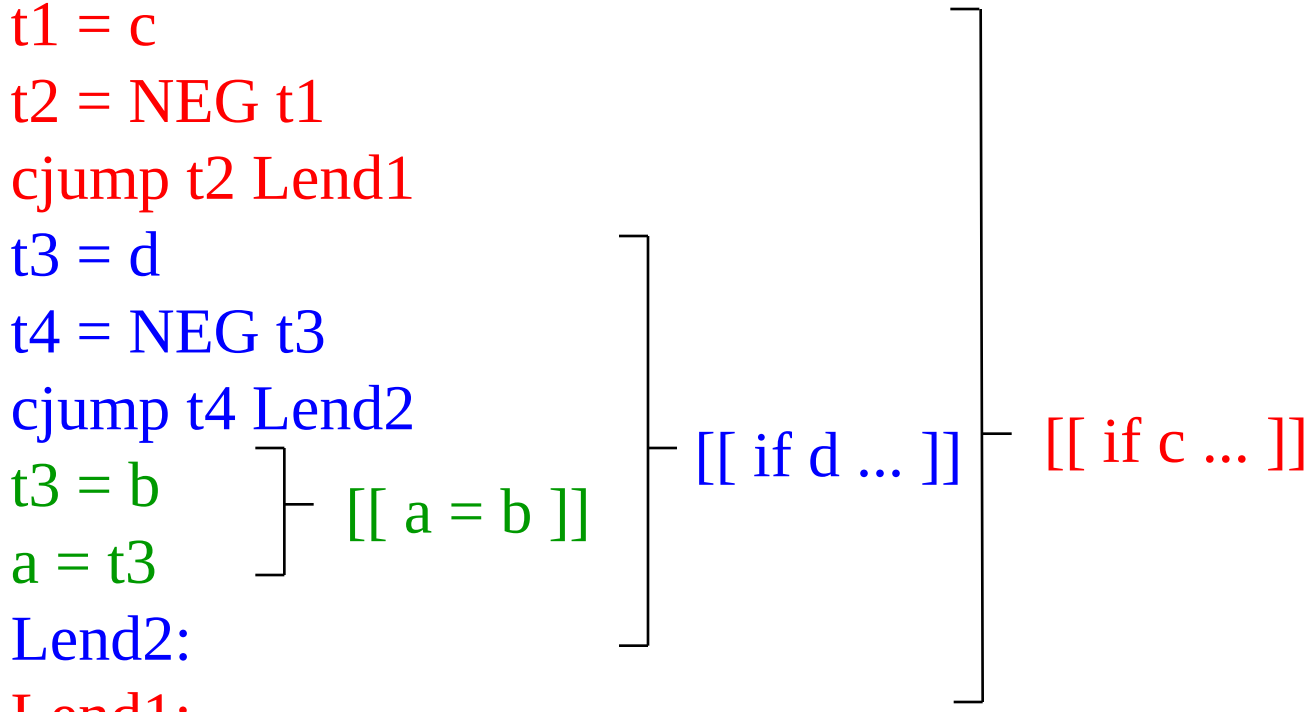
3-주소 Translation 규칙
Nested Expressions

- 중첩된 구조의 expression은 재귀적으로 translate
- Example: t = [[(a – b) * (c + d)]]



3-주소 Translation 규칙
Nested Statements

- 중첩 statement 도 마찬가지
- Example: t = [[if c then if d then a = b]]



Class Problem

다음 코드를 3-주소 코드로 변형하시오.

```
for (i=0; i<100; i++) {
    A[i] = 0;
}
```

```
if ((a > 0) && (b > 0))
    c = 2;
else
    c = 3;
```

3-주소 Translation 규칙 – 추가 팁

- e 이 변수거나 상수일 때

  [[e]] 은 e 이다.

- binary operations: t = [[e1 OP e2]]
  - e1이 변수거나 상수일 때 t2 = [[e2]]
    t = e1 OP t2

  - e2가 변수거나 상수일 때, t1 = [[e1]]
    t = t1 OP e2

- unary operations 의 e1이 변수거나 상수일때 :
  t = [[OP e1]]

  t = OP e1

Note: 다른 식이나 문장에서도 가능한 경우들이 있으나 헷갈리지 않도록 가급적 위와 같은 상황에서만 사용한다.

U-Code Translation

- U-code translation 은
  - 가상 스택 머신에 대한 이해가 필요
  - 즉, storage allocation 등에 대한 이해가 먼저 필요

Storage Process

2 Classes of Storage in Process

- Registers
  - 빠른 접근
  - 일반 프로그래머에게는 보이지 않음
  - 간접 접근 불가
- Memory
  - 상대적으로 느린 접근, 간접접근 가능..
- 변수를 register 로 할 것인지 memory로 할 것인지는 주로 HIR to LIR translation 단계에서 결정됨

Storage Class Selection

- Standard approach
  - Globals/statics – memory
  - Locals
    - Composite types (structs, arrays..) – memory
    - Scalars
      - '&' operator로 접근? – memory
      - 아니면 : 'Virtual' register
- All memory approach
  - 모든 변수를 일단 memory로 하고
  - 이 memory 변수들 중 가능한 것은 register로 조정

Class Problem

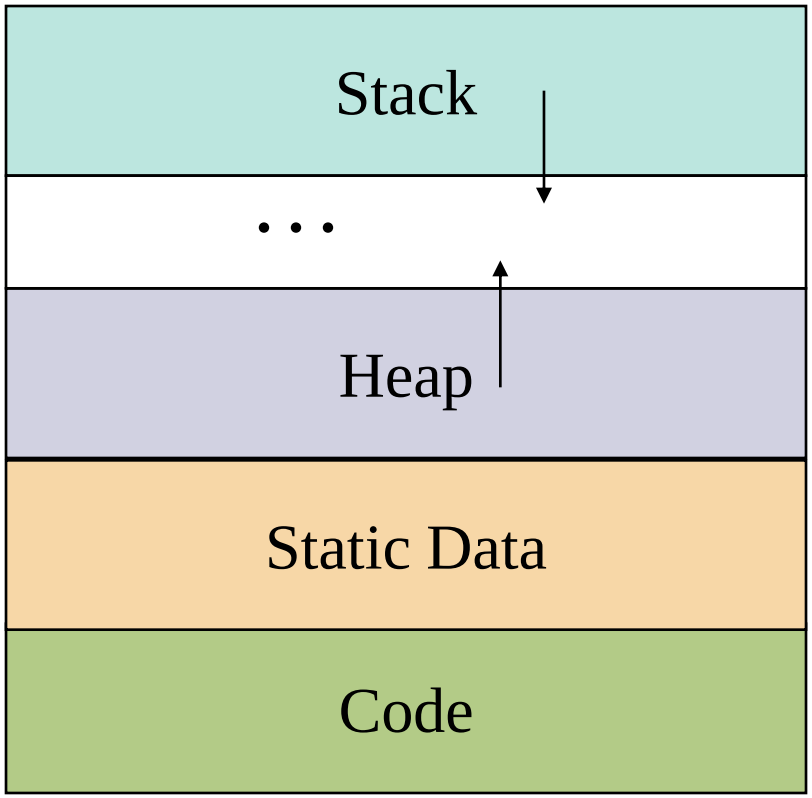
Standard approach 를 썼을 때, 다음 예에서 각 변수가 memory에 저장되는지 register 에 저장되는지 적고, memory에 저장된다면 어떤 영역에 저장되는지 적으시오.

```
int a;
void foo(int b, double c)
{
    int d;
    struct { int e; char f;} g;
    int h[10];
    char i = 5;
    float j;
}
```

Memory의 4대 영역

- Code space : 명령을 저장하는 공간
  - read-only 면 성능 좋음
- Static (or Global) – 프로그램과 life time 을 함께 하는 변수들 집합
- Stack – local 변수들 (블럭 life time)
- Heap – System call (malloc, new) 에 의해 동적으로 allocate 되는 공간

Memory Organization

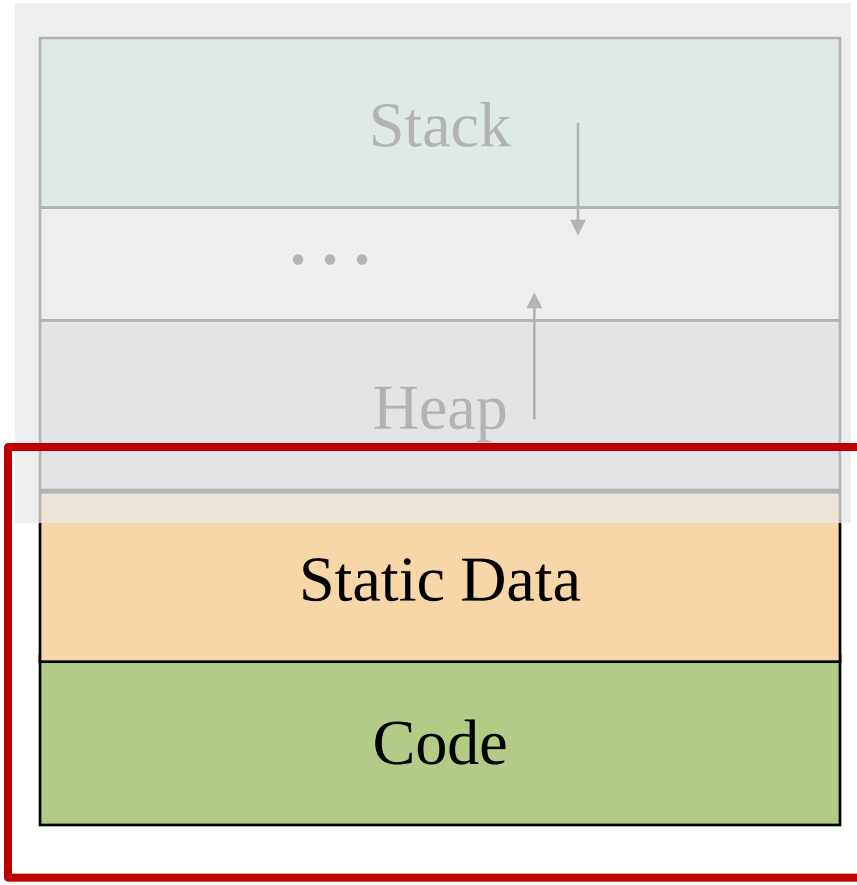


Stack, heap :
런타임에 크기 변함
  Stack 아래쪽으로 증가
  Heap 위로 증가

stack/heap 위치는
반대로 바뀔 수 있음

Code, static data :
컴파일러에 의해 크기 결정

Memory Organization (File Format)



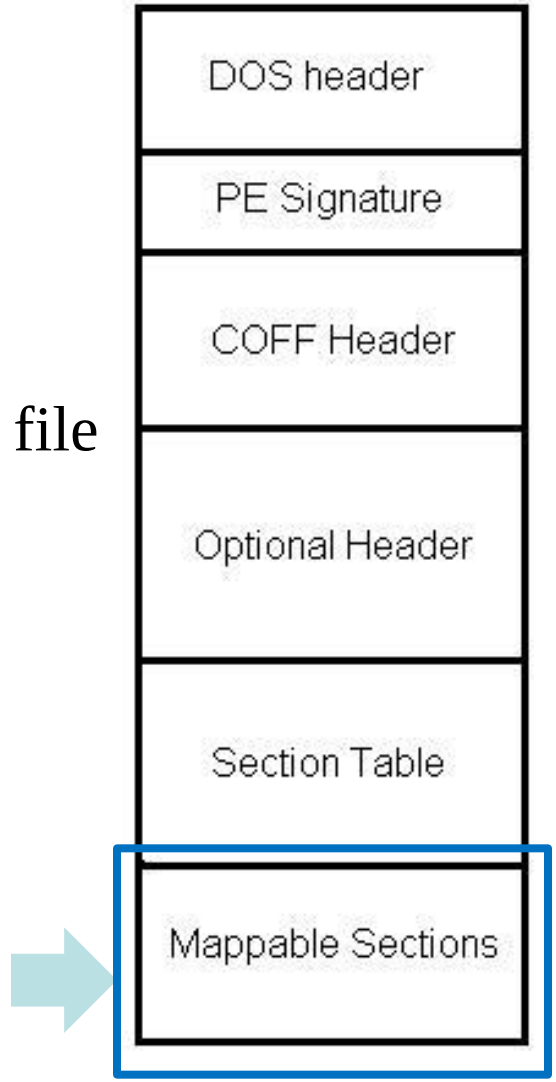
Stack, heap :
런타임에 크기 변함
Stack 아래쪽으로 증가
Heap 위로 증가

stack/heap 위치는
반대로 바뀔 수 있음

Code, static data :
컴파일러에 의해 크기 결정

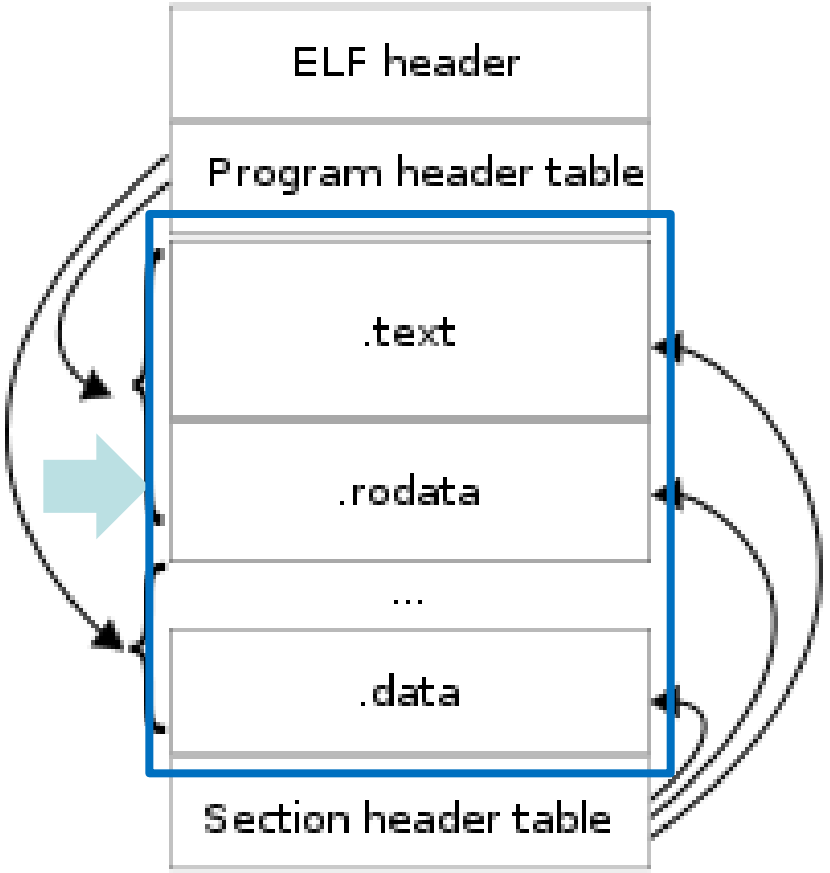
예1) Windows PE (Portable Excutable)

- A file format for executables, object codes and DLLs
- used in 32-bit and 64-bit versions of Windows
- a modified version of the Unix COFF file format

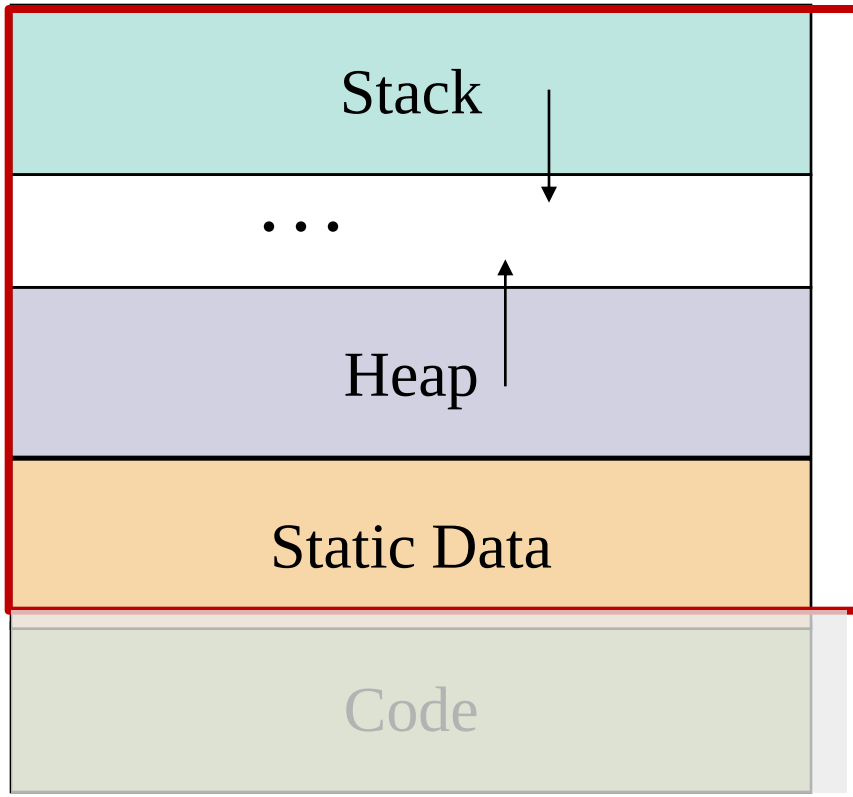


예2) ELF (Executable and Linkable Format)

- A common standard file format for executables, object code, shared libraries, and core dumps
- for Unix-like systems on x86



Memory Organization (Data)



Stack, heap :
런타임에 크기 변함
Stack 아래쪽으로 증가
Heap 위로 증가

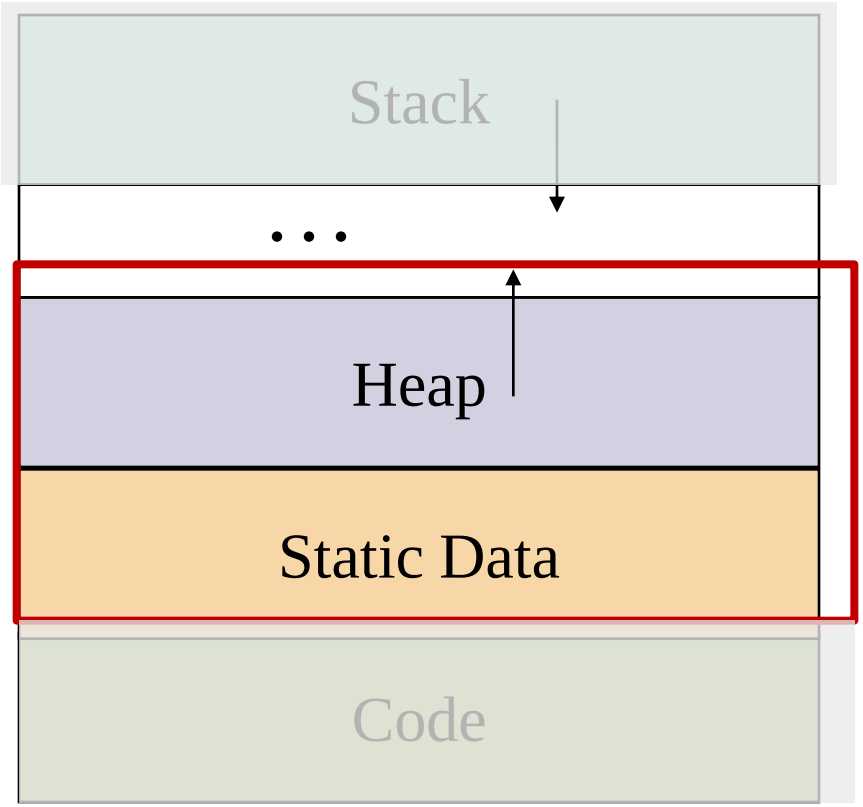
stack/heap 위치는
반대로 바뀔 수 있음

Code, static data :
컴파일러에 의해 크기 결정

변수 바인딩

- 용어
  - environment: <변수, storage location> 정보
    - state : <변수, 값> 정보
  - 바인딩 (binding) :
    - 어떤 environment 에서 변수 이름 N이 storage location S 에 지정되면 N이 S 에 바인딩 된다고 함
    - N이 composite type 이면 여러 (연속적인?) storage location에 바인딩

Memory Organization (Global Data)



Stack, heap :
런타임에 크기 변함
Stack 아래쪽으로 증가
Heap 위로 증가

stack/heap 위치는
반대로 바뀔 수 있음

Code, static data :
컴파일러에 의해 크기 결정

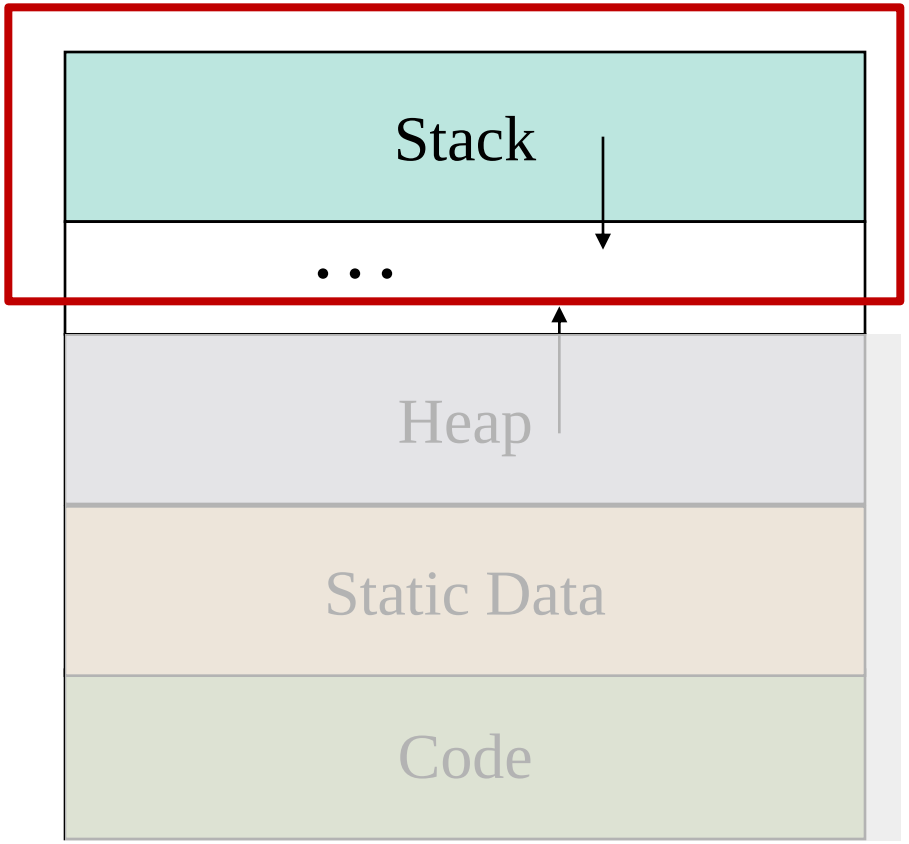
Static Allocation, Heap Allocation

- Static allocation
  - 프로그램 전체 수행 동안 안 변하는 location으로 바인딩 하는 것
  - Used for:
    - Global variables, constants
    - static 변수 in C ➔

```
int count (int n) {
        static int sum = 0;
        sum += n;
}
```

- Heap allocation
  - 연속적인 global 영역의 일부를 OS로 부터 받은 것
  - 프로그램 수행 중 요청과 반환 (deallocate, free) 도 해야함
    - 반환 순서는 무순
  - 안 하려면 garbage collection을 language 에서 지원해야
    - free section, in-use section으로 구분해서 유지 필요 (OS 책참고)

Memory Organization (Stack)



Stack, heap :
런타임에 크기 변함
Stack 아래쪽으로 증가
Heap 위로 증가

stack/heap 위치는
반대로 바뀔 수 있음

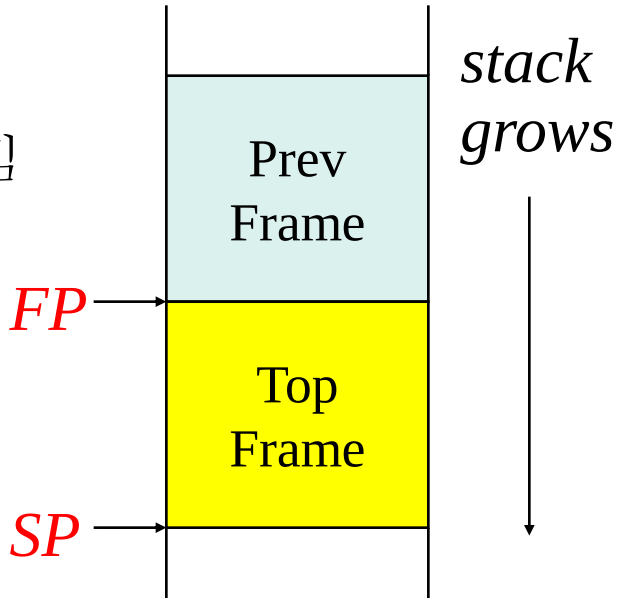
Code, static data :
컴파일러에 의해 크기 결정

Run-Time Stack

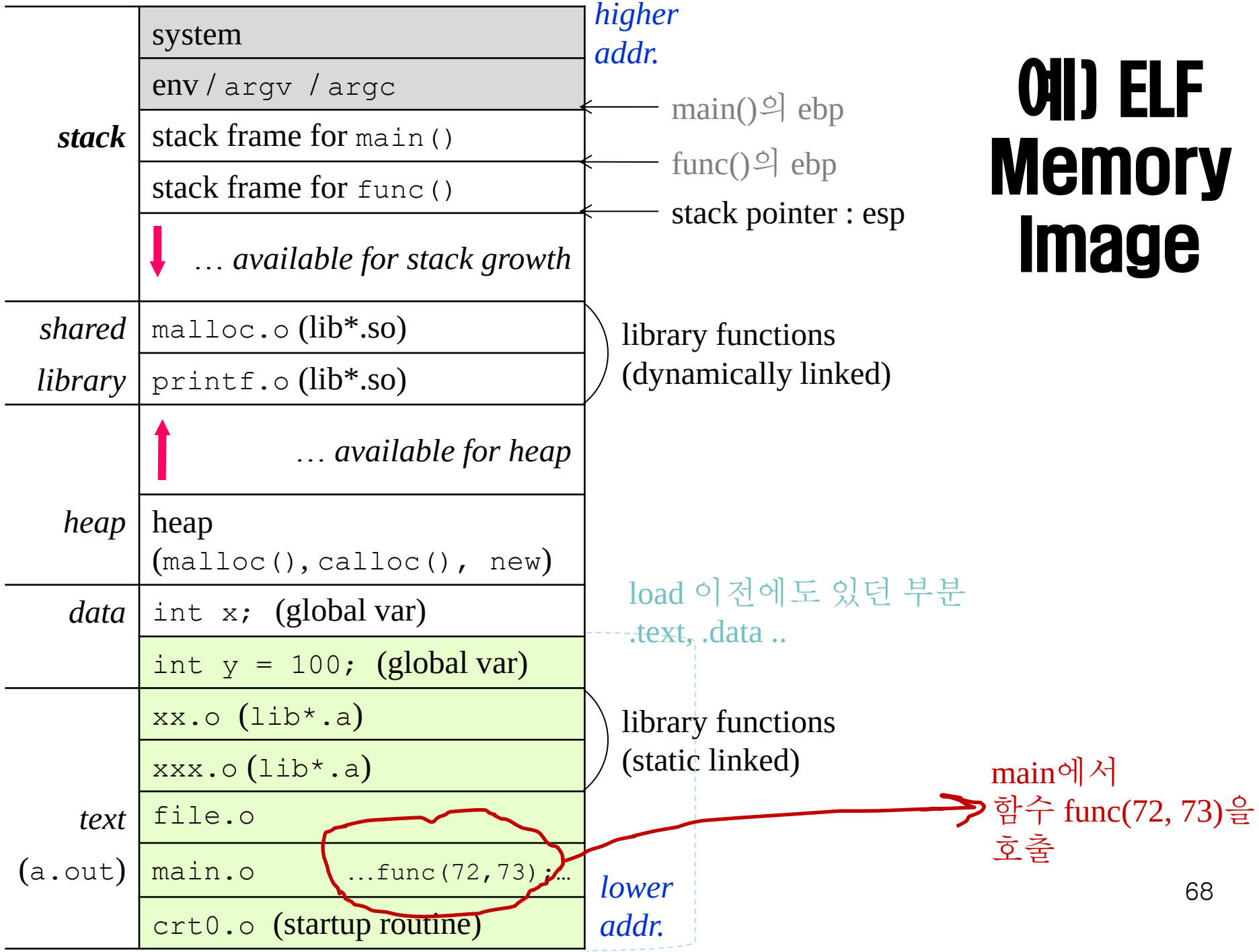
- Run-time stack
  - 한 함수 call 마다 하나씩 두는 frame들 (or activation record) 이 구성하는 stack
- Activation record
  - 해당 함수 수행을 위한 execution environment
    - Recursion 일 때도 각 call 마다 하나씩
    - 내용: local 변수, 인자, 리턴값, 기타 임시 storage ...
- Run-time stack 의 연산
  - f 가 호출되면 f의 frame을 stack에 *push*
  - f 가 return 되면 f 에 대한 frame 을 *pop-up*
  - Top frame = 현재 수행중인 함수의 frame

Stack Pointers

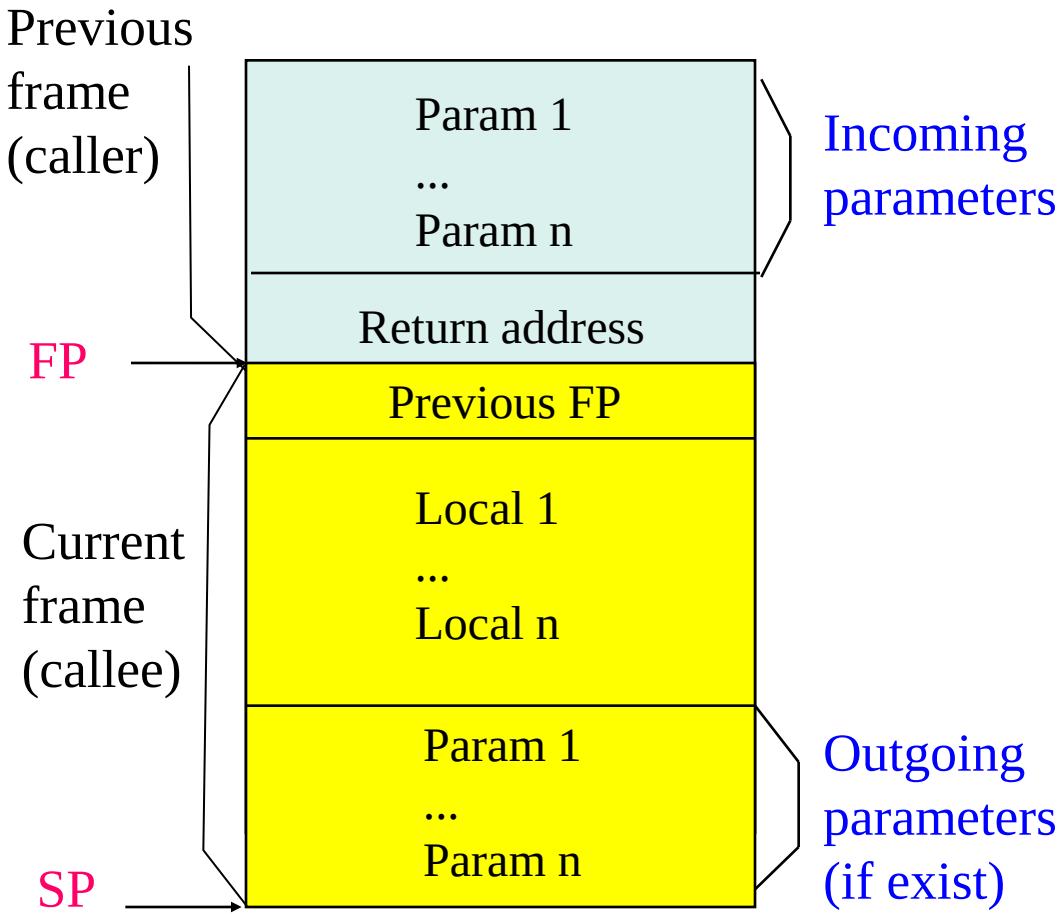
- 일단 stack이 아래로 자란다고 가정하면
  - Stack top의 주소가 수행 중 증가함 (push 시)
- 현재 frame에로의 접근 방법
  - Stack pointer (SP): frame top을 가리킴
  - Frame pointer (FP): frame base를 가리킴
    base pointer라고도 불림
  - 변수는 FP (SP) 로부터의 offset으로 접근
- Why 2 포인터?
  - Small offset 유지
    - instruction 도 짧아짐
  - 사실 수행중 top frame의 절대 위치 모름



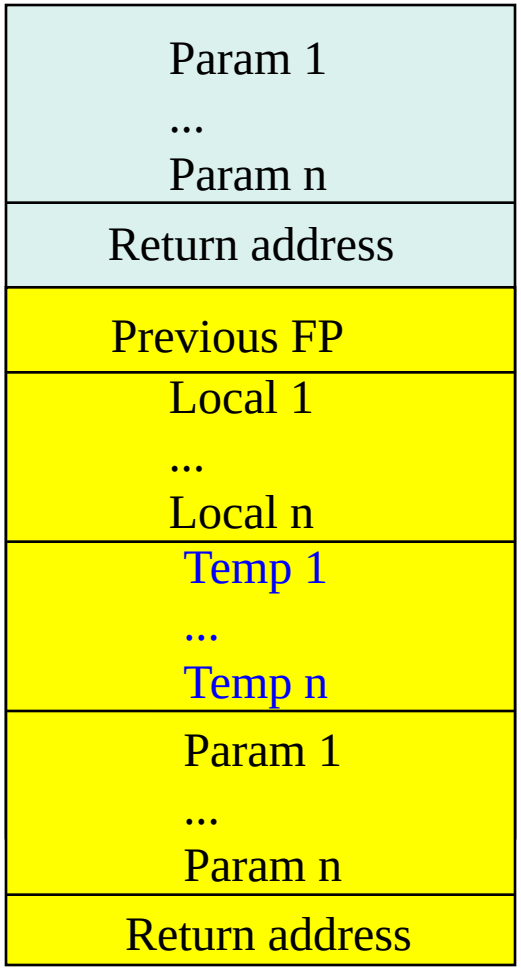
예) ELF Memory Image



Run-Time Stack 해부



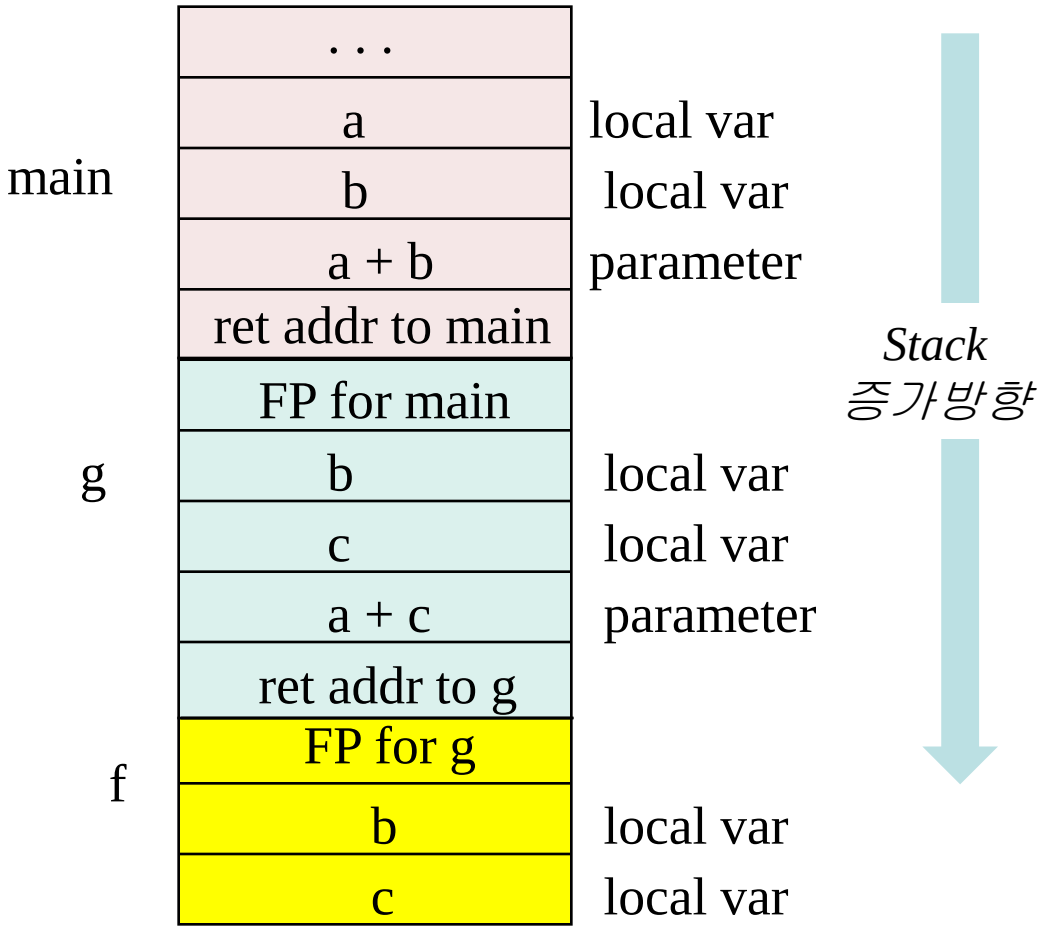
(a) 임시 변수 고려하지 않음



(b) 임시 변수 고려

Stack Frame Construction Example

```
int f(int a) {
    int b, c;
}
void g(int a) {
    int b, c;
    ...
    b = f(a+c);
    ...
}
main() {
    int a, b;
    ...
    g(a+b);
    ...
}
```



Class Problem

다음 프로그램에서

```
void foo(int a) {
    int x=0;
    if (a <= 1) return ;
    foo(a-1);
    foo(a-2);
}

main() {
    int z = 3;
    foo(z);
}
```

1. main에서 시작해서 run-time stack 의 변화 모습을 그려보시오.

2. 프로그램 수행 중에 생성되는 frame의 순간-최대 갯수를 구하시오

함수를 위한 코드 :호출 순서에 따른 설명

- Runtime 스택을 만들고 조작하는 code 를 어떻게 generate 할까? (컴파일러 입장)

- 함수 호출 전에 :
  - actual parameters, *caller-saved registers*, *static link (if necessary)*, 리턴 주소 (current PC) 를 스택에 저장하는 코드 생성
  - callee로 jump하는 코드 생성

- 함수 진입 직전에 (Prolog)
  - FP를 push 하고, old SP 를 new FP로 놓고,
  - *callee-saved registers*와 local 변수를 스택에 push하는 코드를 생성

호출 순서 (계속)

- return 문에서 (Epilog):
  - *callee-saved registers*를 pop(restore)하고,
  - 리턴값을 적절한 장소에 저장하고,
  - old SP, FP 를 restore 하고, (즉 callee frame을 pop)
  - 리턴 주소를 pop하여 그리로 jump하는 코드를 생성

- 함수 호출이 완전히 끝난 후에
  - *caller-saved registers*를 pop(restore) 하고,
  - 리턴 값을 사용하는 코드를 생성

용어

- 함수 호출전 temporary 공간에 register 보관 방법
  - caller 가 보관 하고 함수를 호출하기
  - vs. 함수가 호출된 후 진입 시점에서 callee 가 보관하기
  - vs. 둘다 조금씩 나눠하기
    - Caller-saved registers
    - Callee-saved registers
- Static link
  - dynamic link : old FP 위치를 현 frame에서 보관 (수행 관련)
  - Static link : **nested function** 일 경우, outer function의 frame의 위치를 현 frame에서 보관 (textually nested) (eg. PASCAL, ML, **not C**)

함수 호출 예제

- 가정
  - foo(3,5) 로 호출, foo 함수는 local 변수를 3개 가짐
  - stack 증가 방향은 ↓, 주소 증가방향은 ↑
- 함수 호출 전에
  - push arg1: [sp] = 3
  - push arg2: [sp-4] = 5
  - return address 저장 : [sp-8] = return address
  - 인자 2개, return address 자리 마련하기: sp = sp-12
  - foo 호출 : jump foo
- 함수 진입 직전에 (Prolog)
  - push old FP : [sp] = fp
  - new fp 계산 : fp = sp
  - 3 local (int) 변수 자리마련하기 : sp = sp-4-3*4= sp - 16

함수호출 예제 (계속)

- return 문에서 (Epilog):
  - restore old fp: fp = [fp]
  - pop frame: sp = sp+ 16
  - pop return 주소 and execute return: rts
- 함수 호출이 완전히 끝난 후에
  - 리턴 값 사용하기
  - pop args: sp = sp+ 12

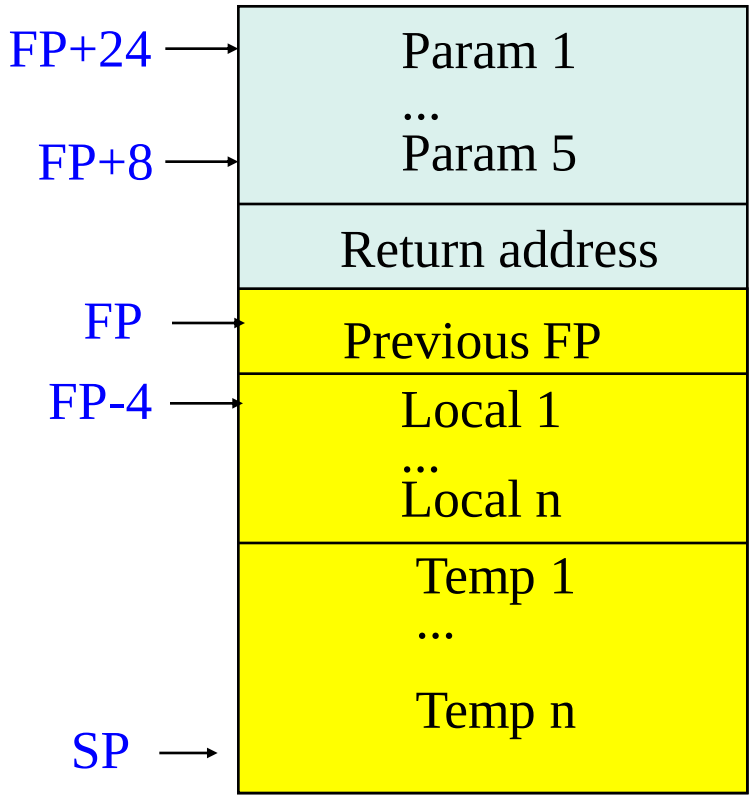
인자 및 변수 접근

- FP로 부터의 offset 으로 stack 접근
- Example
  - stack 증가 방향은 ↓, 주소 증가방향은 ↑ 일 때,

    [fp+8] = param 5

    [fp+24] = param 1

    [fp-4] = local 1



함수를 위한 코드 : 코드 생성 위치

함수 정의 코드 (처리 전)	함수 호출 코드 (처리 전)
함수 body 코드	*… (호출 전 코드들)* *call f(10, 20) ;; 함수호출* *… (호출 후 코드 계속)*

함수를 위한 코드 : 코드 생성 위치

함수 정의 코드 (처리 후)	함수 호출 코드 (처리 후)
Prolog (스택프레임할당) *함수 body 코드* Epilog (스택프레임해제)	… (호출 전 코드들) Parameter, register 등 저장 Return address 저장 Callee로 jump 리턴값 사용 Pop-up args … (호출 후 코드 계속)

함수를 위한 코드 : 코드 생성 위치

함수 정의 코드 (처리 후)	함수 호출 코드(처리 후)
Prolog (스택프레임할당) *함수 body 코드* Epilog (스택프레임해제)	… (호출 전 코드들) (1) Parameter, register 등 저장 Return address 저장 Callee로 jump ← *Return address* 리턴값 사용 Pop-up args … (호출 후 코드 계속)

함수를 위한 코드 : 코드 생성 위치

함수 정의 코드 (처리 후)	함수 호출 코드(처리 후)
	… *(호출 전 코드들)*
(2) Prolog (스택프레임할당)	Parameter, register 등 저장
함수 body 코드	Return address 저장
Epilog (스택프레임해제)	Callee로 jump ← *Return address*
	리턴값 사용
	Pop-up args
	… *(호출 후 코드 계속)*

함수를 위한 코드 : 코드 생성 위치

함수 정의 코드 (처리 후)	함수 호출 코드(처리 후)
Prolog (스택프레임할당)	… (호출 전 코드들)
(3) *함수 body 코드*	Parameter, register 등 저장
Epilog (스택프레임해제)	Return address 저장
	Callee로 jump ← *Return address*
	리턴값 사용
	Pop-up args
	… (호출 후 코드 계속)

함수를 위한 코드 : 코드 생성 위치

함수 정의 코드 (처리 후)	함수 호출 코드(처리 후)
Prolog (스택프레임할당)	… *(호출 전 코드들)*
함수 body 코드	Parameter, register 등 저장
(4) Epilog (스택프레임해제)	Return address 저장
	Callee로 jump ← *Return address*
	리턴값 사용
	Pop-up args
	… *(호출 후 코드 계속)*

함수를 위한 코드 : 코드 생성 위치

함수 정의 코드 (처리 후)	함수 호출 코드(처리 후)
Prolog (스택프레임할당) *함수 body 코드* Epilog (스택프레임해제)	… (호출 전 코드들) Parameter, register 등 저장 Return address 저장 Callee로 jump ← *Return address* 리턴값 사용 (5) Pop-up args … (호출 후 코드 계속)

Class Problem

아래와 같은 코드는 어떤 코드로 변환되겠는가?

즉,

•foo 함수의 시작과 끝은 어떤 코드로 표현되겠는가?

•화살표 부분과 같이 foo 함수를 호출한 부분을 위한 명령은 어떤 코드로 표현되겠는가?

```
int foo(int a, int b) {
    int x, y, z;
    ...
    z = foo(x, y+z);
    ...
    return z;
}
```