Chap 5. Intermediate Representation

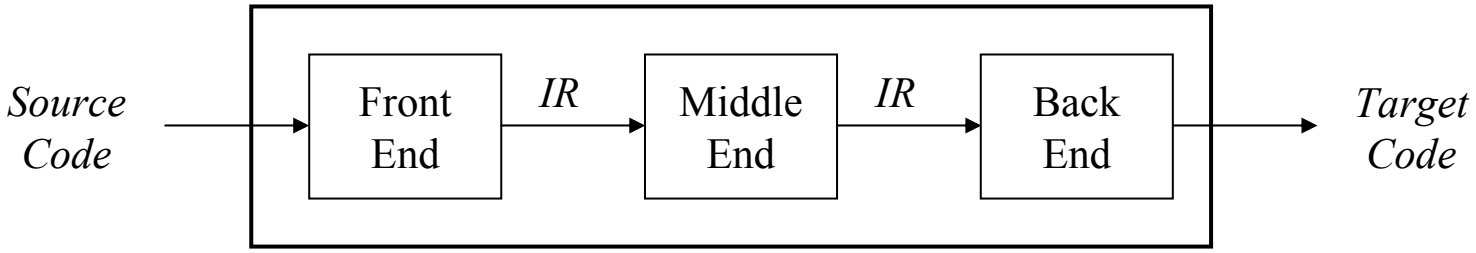
COMP321 컴파일러

2007년 가을학기

경북대학교 전자전기컴퓨터학부

5.1 Introduction

Intermediate Representations



- Front End : produces an **intermediate representation** (*IR*)
- Middle End : transforms the *IR* into an equivalent *IR* that runs more efficiently
- Back End : transforms the *IR* into native code

- *IR* encodes the **compiler's knowledge of the program**
- Middle End usually consists of several passes
  → 여러 개의 *IR*을 사용하기도 함

Intermediate Representations

- *IR*의 선택 → compiler의 speed / efficiency에 영향

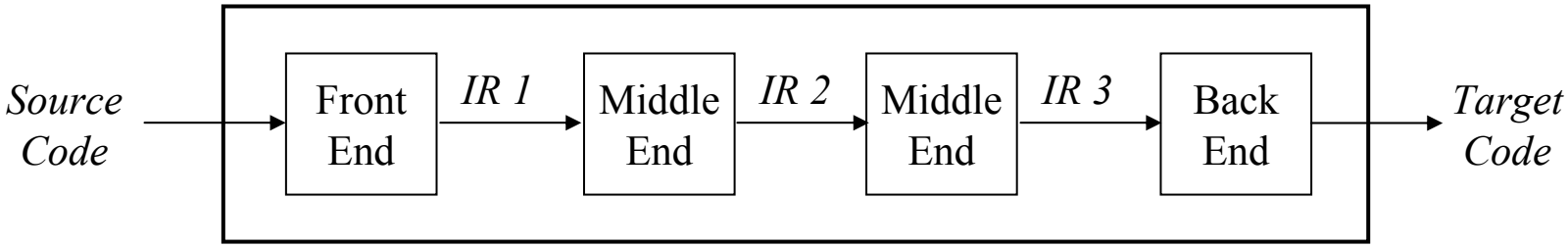
- some important IR properties
  - ease of generation (for Front End)
  - ease of manipulation (for optimization)
  - procedure size
  - freedom of expression
  - level of abstraction
  - clean and readable external format
    - 사람도 이해하기 쉬워야 한다.

IR의 선택

- compiler마다 선택 기준이 다를 수 있음
  - source-to-source translator :
    - 가능한 한 source language에 가까운 *IR* 사용
  - traditional compiler:
    - 가능한 한 assembly language에 가까운 *IR* 사용

- Selecting an appropriate IR for a compiler is critical

Using Multiple Representations



- **Repeatedly lower the level** of the intermediate representation
  - Each intermediate representation is suited towards certain optimizations
- Example: the Open64 compiler
  - WHIRL intermediate format
    - Consists of **5 different *IR*s** that are progressively more detailed

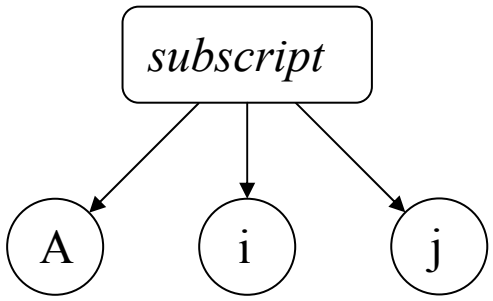
5.2 Taxonomy

IR의 분류

- **Graphical *IR*'s**
  - tree 나 graph (DAG) 형태
    - node, edge, list, tree로 표현 → size가 커짐
  - heavily used in source-to-source translators
- **Linear *IR*'s**
  - pseudo-code for an abstract machine
  - simple, compact data structures
  - easier to rearrange
- **Hybrid *IR*'s**
  - linear IR로 각 block을 표시, block끼리는 graphical IR
  - example: CFG (control flow graph)

Level of Abstraction에 의한 분류

- level of abstraction in *IR* :
  - profitability and feasibility of different optimizations.
- example: two *IR*'s of an array reference $A[i, j]$



Source-level AST:
good for memory
usage check

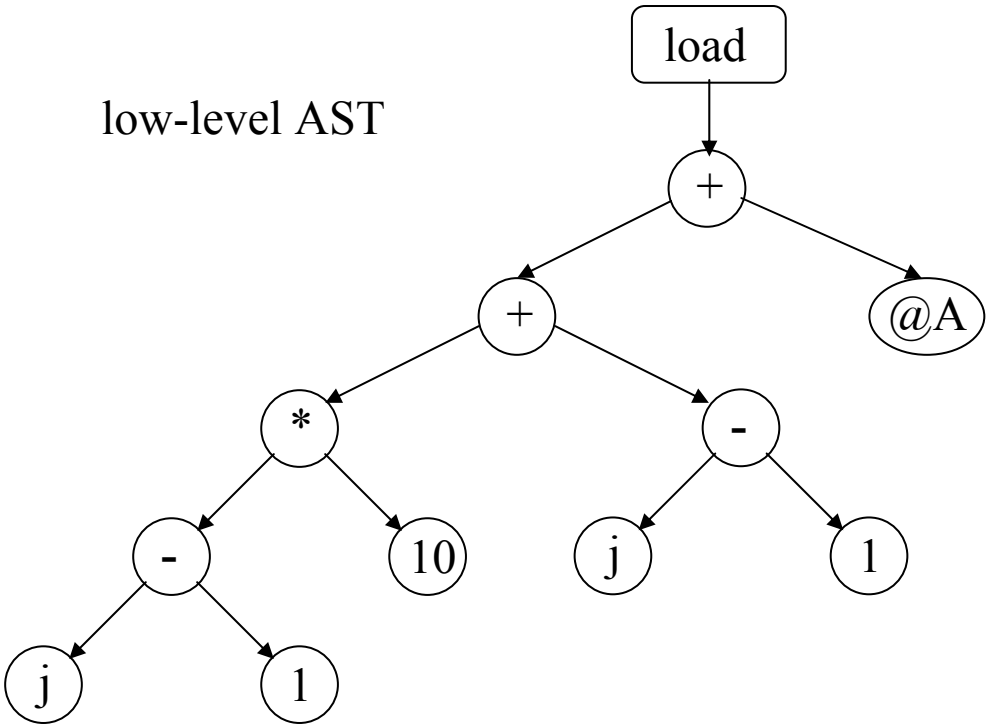
```
loadI  1      ⇒ r1
sub    rj, r1 ⇒ r2
loadI  10     ⇒ r3
mult   r2, r3 ⇒ r4
sub    ri, r1 ⇒ r5
add    r4, r5 ⇒ r6
loadI  @A     ⇒ r7
Add    r7, r6 ⇒ r8
load   r8     ⇒ rAij
```

Low level linear code:
Good for address calculation

Level of Abstraction에 의한 분류

- Graphical *IR*'s : usually high-level
- Linear *IR*'s : usually low-level
  - not necessarily true:

low-level AST



loadArray A, i, j

high-level linear code :
나중에 확장하거나,
system call 또는
library call로 대체 가능

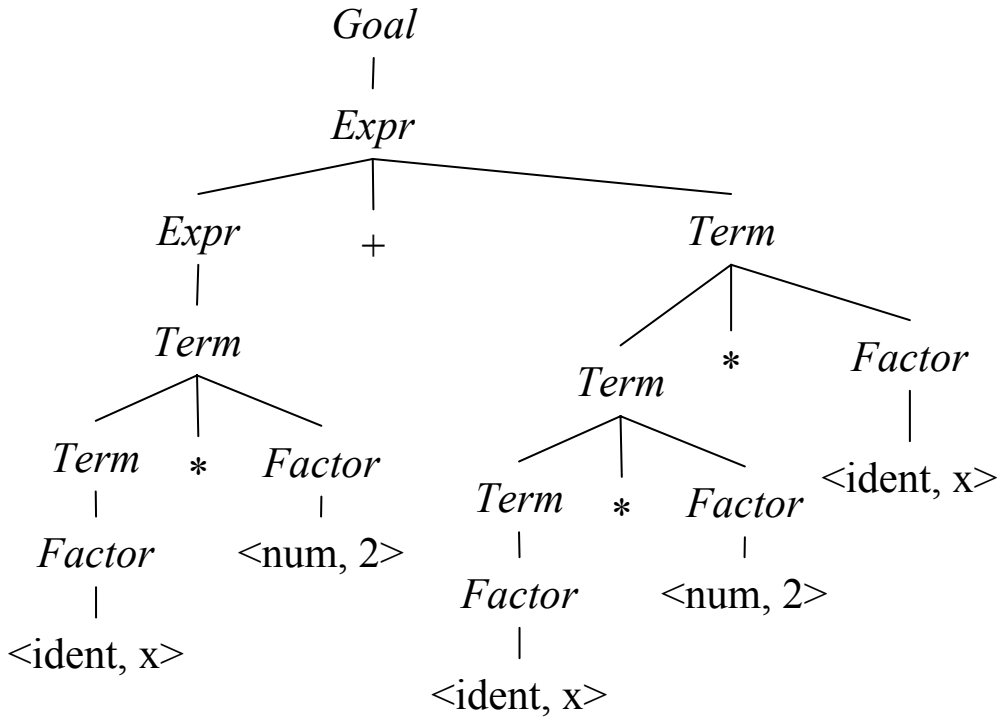
5.3 Graphical IRs

Parse Trees

- **parse tree** 자체를 그대로 *IR* 로 사용 가능
  - 문제점: 불필요한 node가 많다.

Goal → *Expr*
Expr → *Expr* + *Term*
| *Expr* – *Term*
| *Term*
Term → *Term* * *Factor*
| *Term* / *Factor*
| *Factor*
Factor → (*Expr*)
| **num**
| **ident**

x * 2 + x * 2 * y

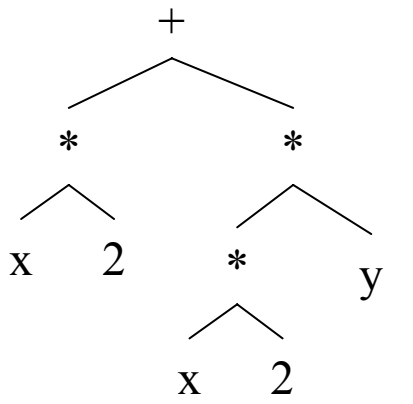


AST: Abstract Syntax Tree

- 불필요한 **node**가 제거된 **parse tree**
  - 주로 non-terminal node들을 제거

- near source-level representation
  - source code의 pretty-printing 또는 regeneration에 사용
  - postfix: x 2 * x 2 * y * +

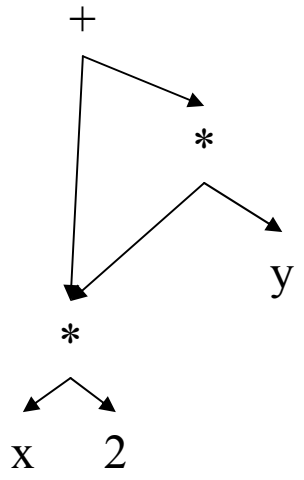
x * 2 + x * 2 * y



DAG: Directed Acyclic Graph

- **AST에서 duplication을 제거한 형태**
  - tree에서 graph로 바뀜

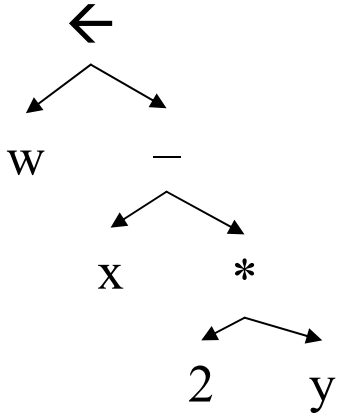
x * 2 + x * 2 * y



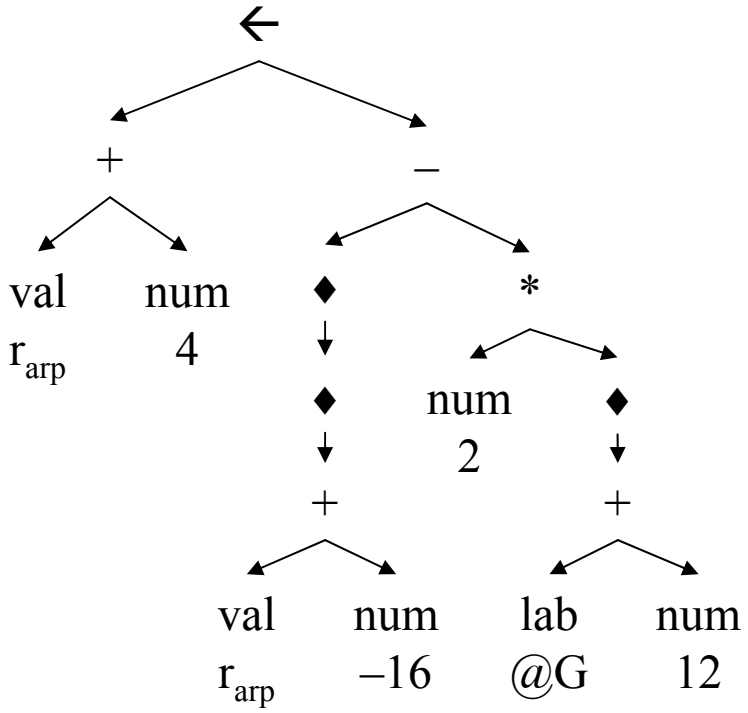
- Makes sharing explicit
- Encodes redundancy
  - Same expression twice
    means that the compiler might
    arrange to evaluate it just once!

Level of Abstraction Again

- AST나 DAG에서 **level**에 따라 다른 표현이 가능
- example: w ← x – 2 * y



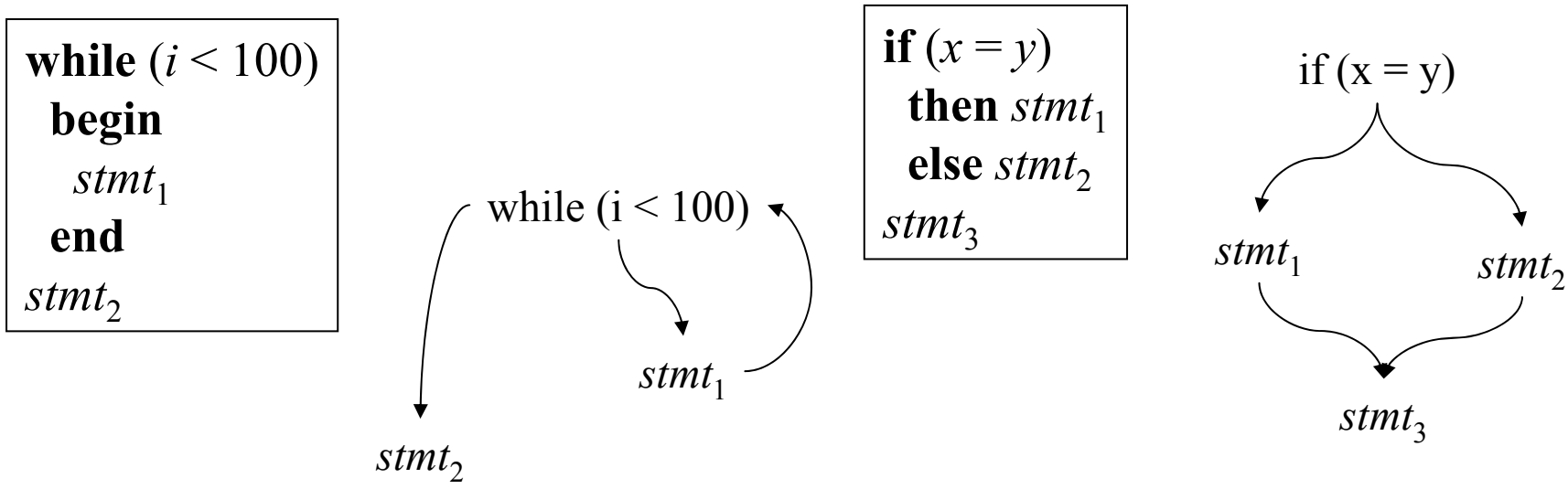
source-level AST



low-level AST
memory address 계산을 포함

CFG: Control-Flow Graph

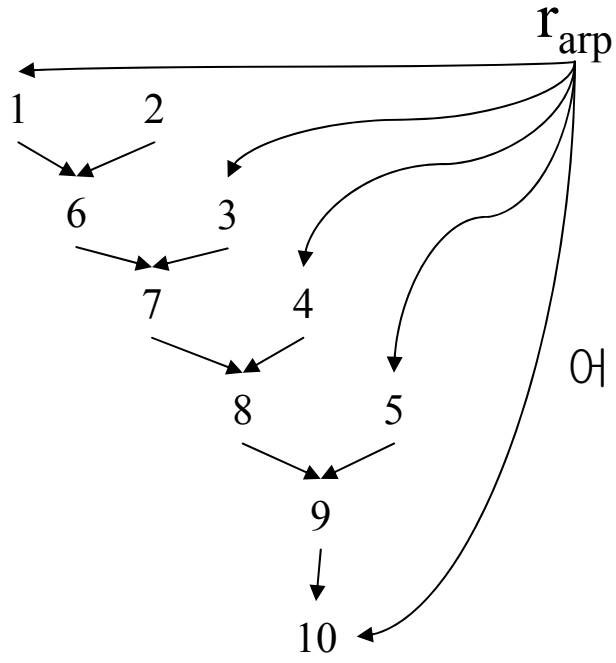
- **flow of control 을 modeling**
  - node : a basic block
  - edge : block에서 block 으로의 control transfer



Dependence Graph

- **order-of-evaluation**을 정하기 위해서 사용
  - definition point와 use point를 서로 연결
  - 주로 linear IR에서 optimize 용으로 사용

1	loadAI r_{arp}, @w => r_w
2	loadI 2 => r_2
3	loadAI r_{arp}, @x => r_x
4	loadAI r_{arp}, @y => r_y
5	loadAI r_{arp}, @z => r_z
6	mult r_w, r_2 => r_w
7	mult r_w, r_x => r_w
8	mult r_w, r_z => r_w
9	mult r_w, r_z => r_w
10	storeAI r_w => r_{arp}, 0



어디서 사용되는가?

5.4 Linear IRs

Linear *IR*

- **assembly language**에 가까운 형태
  - should be executed in their order of appearance

- 종류
  - one-address codes : accumulator model
    - C ← A + B : load A, add B store C
  - two-address codes : register model
    - C ← A + B : load r1 A, add r1 B, store C r1
  - three-address codes : many register model
    - C ← A + B : add r_C, r_A, r_B

Stack Machine Code

- **one-address code**라고도 함.
  - 원래 stack-based machine용, 현재는 Java VM
  - example: x – 2 * y

```
push   2
push   y
multiply
push   x
subtract
```

- Advantages
  - compact form
    - network 전송에 유리 !
  - introduced names are implicit, not explicit
  - simple to generate and execute code

Three-Address Code

- 여러 가지 형식 가능
  - x ← y op z
  - 1 operator (op) and, at most, 3 names (x, y, z)
  - example: x – 2 * y

$$
\begin{aligned}
t_1 &\leftarrow 2 \\
t_2 &\leftarrow y \\
t_3 &\leftarrow t_1 * t_2 \\
t_4 &\leftarrow x \\
t_5 &\leftarrow t_4 - t_1
\end{aligned}
$$

- Advantages:
  - **resembles many machines**
  - introduces a new set of names
  - compact form

Three-Address Code: Quadruple 표현

- three-address code를 표현하는 naïve한 방법
  - array of *k* * 4 small integers
  - Original ForTran에서 사용

```
load  r1, y
loadI r2, 2
mult  r3, r2, r1
load  r4, x
sub   r5, r4, r3
```

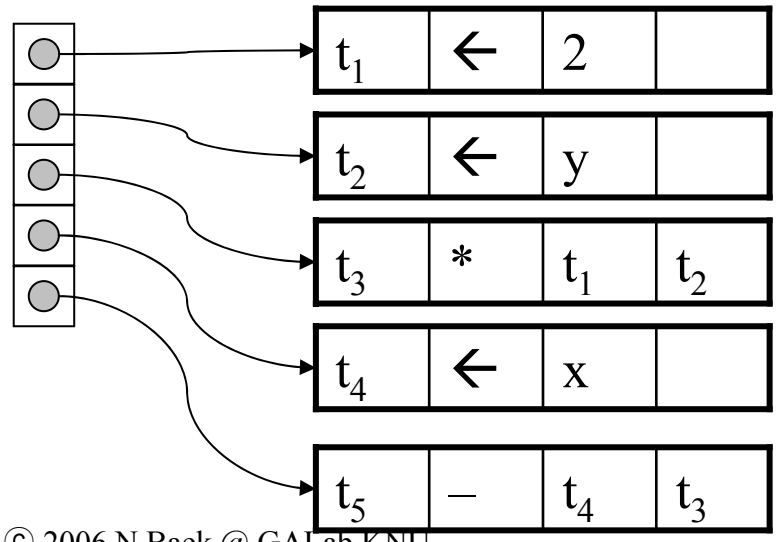
RISC assembly code

load	1	Y	
loadi	2	2	
mult	3	2	1
load	4	X	
sub	5	4	2

Quadruples
(text에는 다른 방식 표현)

Three-Address Code: list 표현

- quadruple 방식의 단점: 재배치하기 어려움
  - optimizing 시에 더 좋은 방법?
- array of pointers
- linked list
  - 모두 pointer를 이용해서 quadruple의 재배치



5.5 Static Single-Assignment Form

SSA Form

- **static single-assignment form**
  - control flow 와 data flow를 추가하는 한 방법
  - main idea : each name defined exactly once

```
x ← …
y ← …
while (x < 100)
  x ← x + 1
  y ← y + x
end while
…
```

original code
x 끼리의 구별은?

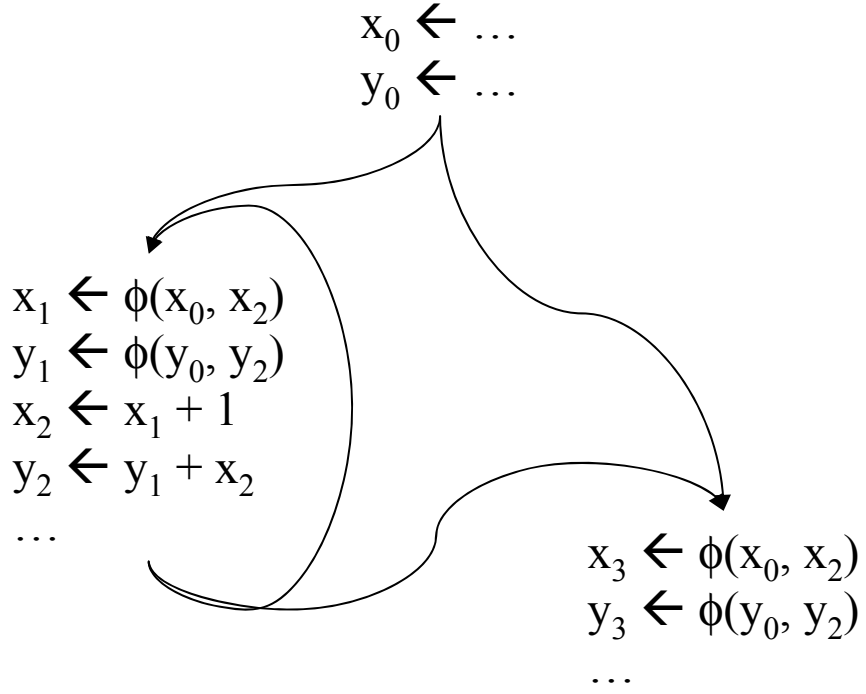
```
        x0 ← …
        y0 ← …
        if (x0 ≥ 100) goto next
loop:   x1 ← φ(x0, x2)
        y1 ← φ(y0, y2)
        x2 ← x1 + 1
        y2 ← y1 + x2
        if (x2 < 100) goto loop
next:   x3 ← φ(x0, x2)
        y3 ← φ(y0, y2)
        …
```

SSA form

SSA Form

- code optimization에서 유용하게 사용
- **ϕ function**: control flow에 따라, 택하는 data가 다르다.

```
            x0 ← …
            y0 ← …
            if (x0 ≥ 100) goto next
loop:       x1 ← ϕ(x0, x2)
            y1 ← ϕ(y0, y2)
            x2 ← x1 + 1
            y2 ← y1 + x2
            if (x2 < 100) goto loop
next:       x3 ← ϕ(x0, x2)
            y3 ← ϕ(y0, y2)
            …
```



5.6 Mapping Values to Names

Mapping Values to Names

- source code
  - 착각하기 쉽다.

  a ← b + c

  b ← a – d

  c ← b + c

  d ← a – d

a, c는 다른 값 : 같은 값으로 착각하기 쉽다.

b, d는 같은 값 : 눈에 잘 띄지 않는다.

- using **value names**
  - 착각을 없앤다.
  - optimization에 유리

  t1 ← b

  t2 ← c

  t3 ← t1 + t2

  a ← t3

  t4 ← d

  t5 ← t3 – t4

  b ← t5

  t6 ← t5 + t2

  c ← t6

  t5 ← t3 – t4

  d ← t5

Memory Models

- **Register-to-register model**
  - 모든 변수가 register에 저장된다고 가정
  - register 개수에 제한 없이 사용
  - **compiler back-end에서 (real) load / store 추가**
  - RISC machine에서 주로 사용

- **Memory-to-memory model**
  - 모든 변수가 memory에 저장된다고 가정
  - **compiler back-end에서 불필요한 load / store 제거**

5.6 Symbol Tables

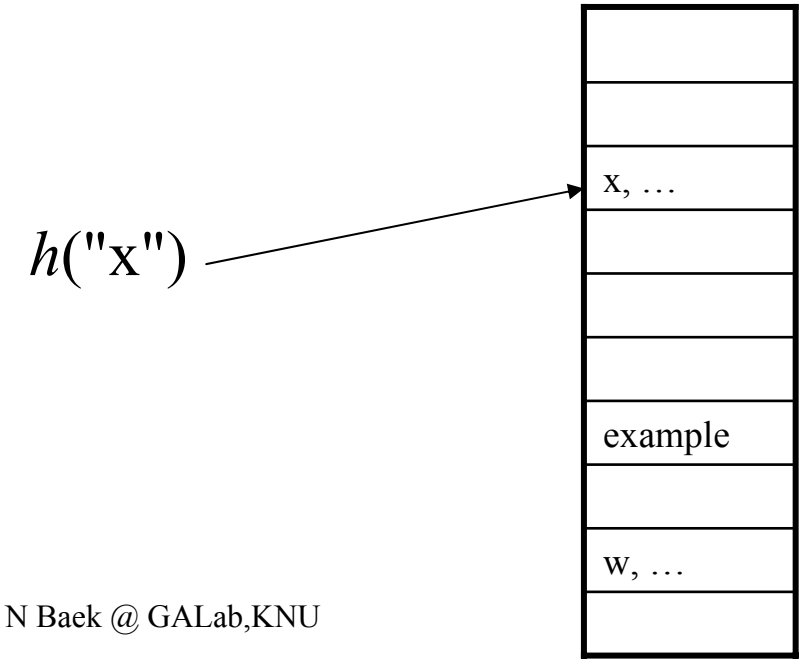
Compiler가 필요로 하는 정보들

- names for
  - variables, defined constants, procedures, functions, labels, structures, files, …
- extra information for variables
  - data type, storage class, name, memory address, …

- 이 정보들을 어디엔가 저장해야 한다.
  - in AST nodes : redundancy 발생
  - 해결책 : **별도의 symbol table 관리**
    - efficiency 가 중요 : compiler의 모든 단계에서 사용

Hash Tables

- symbol table 구성
  - efficiency : hash table로 해야 달성 가능
    - 항상 constant time에 정보를 돌려 줌
  - 가정: $h(n)$ is a perfect hash function where n is the symbol name.



Symbol Table의 기본 연산

- **LookUp(*name*)**
  - h(name) 에 대응되는 정보가 있으면, name을 가지는 variable / function 등에 대한 정보를 가져옴.
  - 없으면, not found
- **Insert(*name*, *record*)**
  - h(name) 에 대응되는 slot에 record에 담긴 정보를 저장.
- 필요하면, 다른 연산도 추가할 것
  - initialize, finalize, …

Nested Scope

- a single unified name space : BASIC의 경우
  - symbol table 1개로 처리 가능
- nested scope languages : C, C++, Java, …
  - scope에 따른 처리 필요

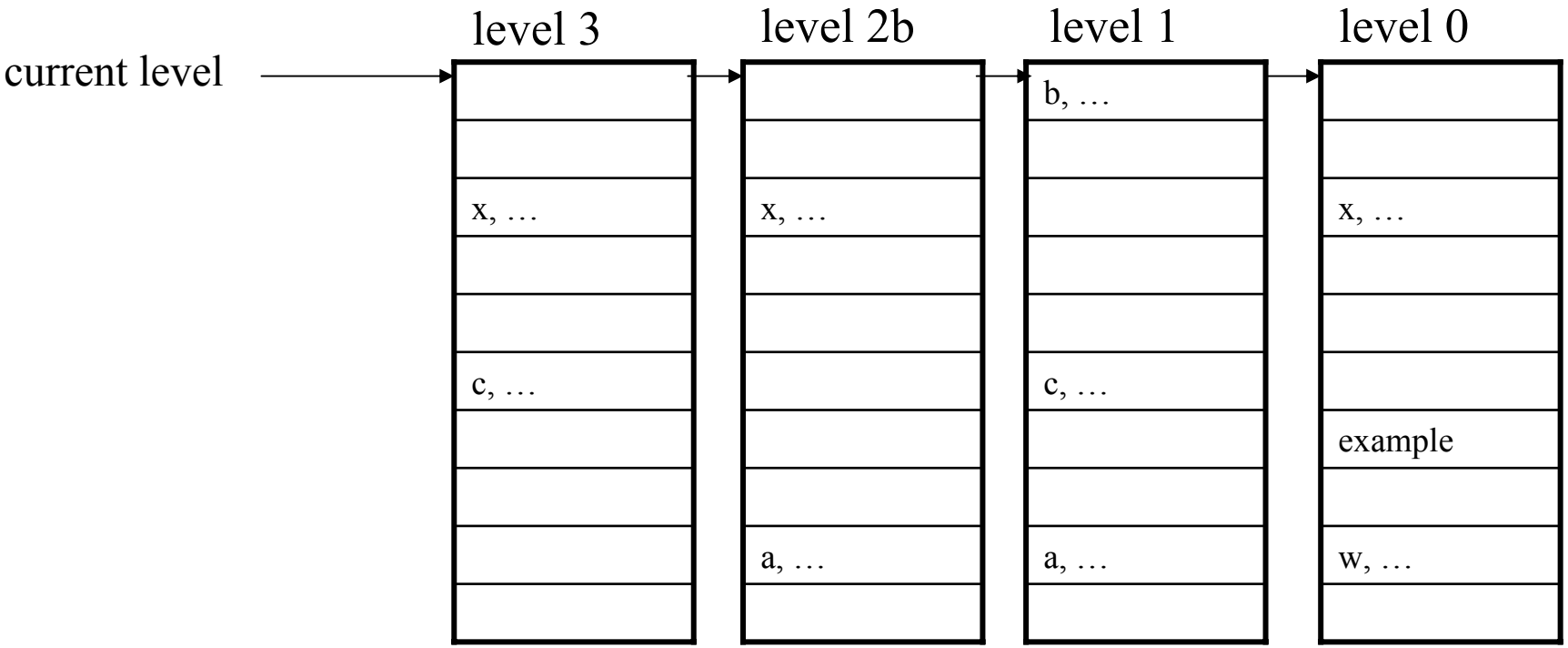
```
static int w;        /* level 0 */
void example(int a, int b) {
   int c;            /* level 1 */
   {  int b, z;      /* level 2a */
      …
   }
   {  int a, x;      /* level 2b */
      …
      { int c, x;    /* level 3 */
        b = a + b + c + w;
        …
```

level	name
0	w, example( )
1	a, b, c
2a	b, z
2b	a, x
3	c, x

$b_1 = a_{2b} + b_1 + c_3 + w_0$

Nested Scope 처리

- scope 별 symbol table 유지
  - InitializeScope( )
  - FinalizeScope( )



Structure의 처리

- structure : 내부에 field를 가진다.
  - field name : variable name과는 구별 필요
- symbol table에서의 처리 방법?
  - **separate tables** : structure 마다 별도의 symbol table
  - **selector table** : structure는 main symbol table에 저장,
    모든 field는 별도의 symbol table에 저장
    - 같은 field name : 충돌 방지 방법 필요
  - **unified table** : 모든 field를 main symbol table에 저장
    - 각 field를 구별해야 함
      → 별도의 qualified naming 방법 필요

OOPL 에서의 추가 처리

- object oriented programming language에서는
  - lexical scope : 당연히 필요
  - **class hierarchy** : 추가적인 scope로 작용
    - derived class는 **base class**의 **field**를 사용 가능
  - 결국, 더 복잡한 symbol table 관리가 필요