Chap 7. Code Shape

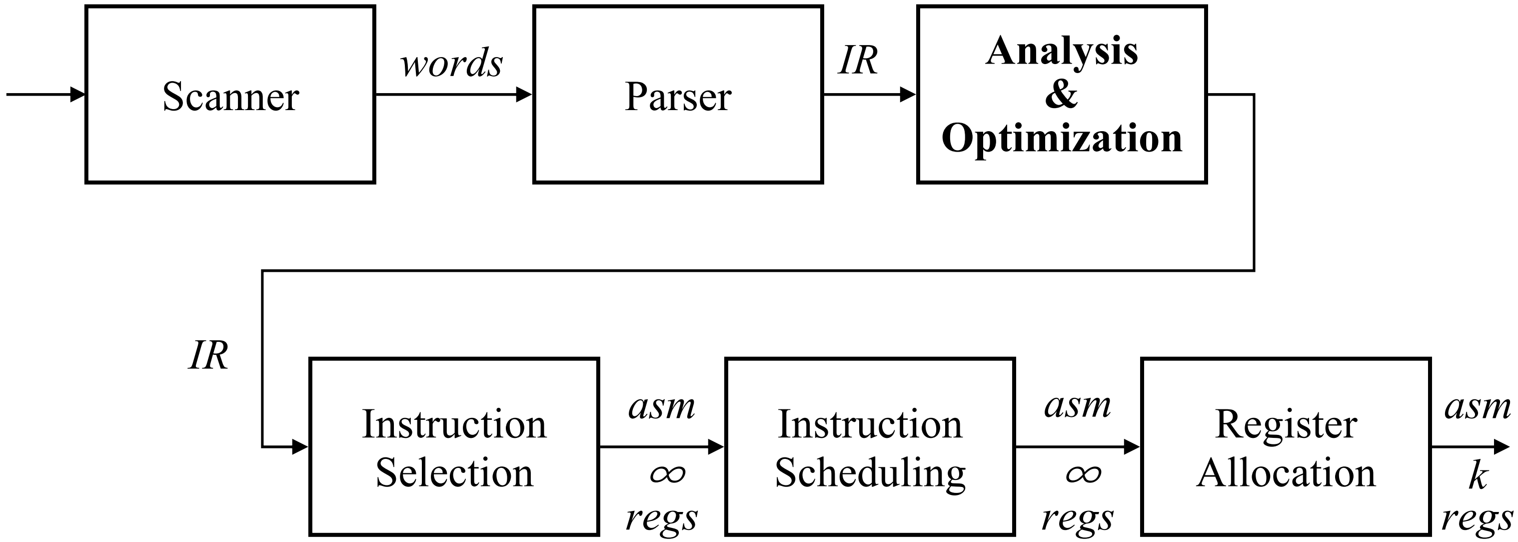
COMP321 컴파일러

2007년 가을학기

경북대학교 전자전기컴퓨터학부

7.1 Code Shape

Structure of a Compiler



What about the IR ?

- Low-level, **RISC-like IR called ILOC**
- Has "enough" registers
- ILOC was designed for this stuff

Definitions

- **Instruction Selection**
  - Mapping: **IR → assembly code**
  - fixed storage로 mapping 하고, 실제 code 생성
  - 어느 operation ? 어떤 addressing mode ?
- **Instruction Scheduling**
  - **Reordering** operations to hide latencies
  - 결과적으로 필요한 register 수를 줄임
- **Register Allocation**
  - 어느 register에 어떤 값을 넣을 것인가 ?
  - memory ←→ register 간의 이동

Code Shape

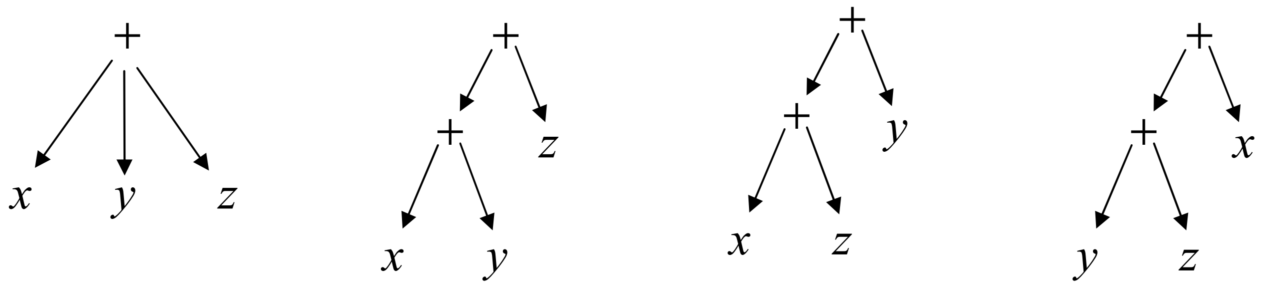
- the same source → many different execution files
  - compiler 마다 **서로 다른 code**를 생성
  - 이런 code 생성 시의 차이가 code shape

- example: switch with **0 ~ 255 cases in C**
  - plain if-then-else 구현: 평균 128번의 if test
  - binary search: 평균 8번의 if test
  - lookup table with 256 entries: 가장 빠름
    → compiler 마다 다른 선택을 한다.

Code Shape

- another example

$x + y + z$	$x + y \rightarrow t1$	$x + z \rightarrow t1$	$y + z \rightarrow t1$
	$t1 + z \rightarrow t2$	$t1 + y \rightarrow t2$	$t1 + z \rightarrow t2$



- **What is the best choice ?**
  - when $y = 2$ and $z = 3$?
  - when we already have $w = x + y$?
- contextual knowledge 필요 !

7.2 Assigning Storage Locations

Memory & Register

- **memory model**
  - procedure local variables → activation record 영역
  - procedure static variables → global data 영역
  - global variables → global data 영역
  - memory allocation → run-time heap 영역

- **register model**
  - register에 variable을 두면, 빠르고, 편리하다.
  - 제약 조건 : register의 address는 없다 !

Variable의 분류

- **unambiguous value**
  - **register에 variable을 assign 할 수 있는 조건**
    - address를 구하지 않는다. (value만 사용한다)
    - nested procedure에서 사용하지 않는다.
    - call-by-reference에서 사용하지 않는다.
  - → 어느 경우든, 위배하면, ambiguous value로 판정
- **ambiguous value**
  - variable 접근 방법이 여러 개 있는 경우

Machine Specific Rules

- CPU / Memory 마다 지킬 rule 이 있다.
  - **register 용도**가 정해져 있을 수 있다.
    - SP : stack pointer, BP: base pointer, …
  - register 몇 개가 묶여서 사용될 수 있다.
    - AL, AH : 각각 8-bit general register
    - AX = (AH, AL) : 16-bit general register
  - 모든 data는 **word boundary**에 맞추어야 한다.
    - 16-bit → 32-bit → 64-bit 로 확대 중.
    - 8-bit data는?   dummy byte를 붙여서 맞춘다

7.3 Arithmetic Operators

Code Generation for Expressions

- start point of the code generation
- to evaluate x + y:

  loadI @x $\Rightarrow r_1$

  loadAO $r_{arp}, r_1 \Rightarrow r_x$

  loadI @y $\Rightarrow r_2$

  loadAO $r_{arp}, r_2 \Rightarrow r_y$

  add $r_x, r_y \Rightarrow r_t$
- how to automate it ?
  - recursion process !

Code Generation for Expressions

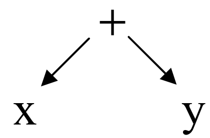
```
expr(node) {
  int result, t1, t2;
  switch (type(node)) {
      case ×,÷,+,– :
          t1← expr(left child(node));
          t2← expr(right child(node));
          result ← NextRegister( );
          emit(op(node), t1, t2, result);
          break;
      case IDENTIFIER:
          t1← base(node);
          t2← offset(node);
          result ← NextRegister( );
          emit(loadAO, t1, t2, result);
          break;
      case NUMBER:
          result ← NextRegister( );
          emit(loadI, val(node), none, result);
          break;
      }
      return result;
}
```

- The concept
  - Use **a simple treewalk evaluator**
  - 복잡한 것은 모두 function으로
    - *base*(), *offset*(), & *val*()
- implements expected behavior
  - Visits & evaluates **children**
  - Emits code for the **op itself**
  - Returns register with result
- simple expression 에서 잘 작동 !
  - Easily extended to other operators
  - Does not handle control flow

Code Generation for Expressions

```
expr(node) {
  int result, t1, t2;
  switch (type(node)) {
      case ×,÷,+,– :
          t1← expr(left child(node));
          t2← expr(right child(node));
          result ← NextRegister( );
          emit(op(node), t1, t2, result);
          break;
      case IDENTIFIER:
          t1← base(node);
          t2← offset(node);
          result ← NextRegister( );
          emit(loadAO, t1, t2, result);
          break;
      case NUMBER:
          result ← NextRegister( );
          emit(loadI, val(node), none, result);
          break;
      }
      return result;
}
```

Example:



Produces:

expr("x") →

loadI	**@x**	⇒ **r1**
loadAO	**r0, r1**	⇒ **r2**

expr("y") →

loadI	**@y**	⇒ **r3**
loadAO	**r0, r3**	⇒ **r4**

NextRegister() → r5

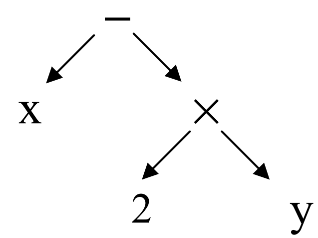
emit(add,r2,r4,r5) →

add	**r2, r4**	⇒ **r5**

Code Generation for Expressions

```
expr(node) {
  int result, t1, t2;
  switch (type(node)) {
      case ×,÷,+,– :
          t1← expr(left child(node));
          t2← expr(right child(node));
          result ← NextRegister( );
          emit(op(node), t1, t2, result);
          break;
      case IDENTIFIER:
          t1← base(node);
          t2← offset(node);
          result ← NextRegister( );
          emit(loadAO, t1, t2, result);
          break;
      case NUMBER:
          result ← NextRegister( );
          emit(loadI, val(node), none, result);
          break;
   }
   return result;
}
```

Example:



Generates:

loadI	**@x**	**⇒ r1**
loadAO	**r0, r1**	**⇒ r2**
loadI	**2**	**⇒ r3**
loadI	**@y**	**⇒ r4**
loadAO	**r0,r4**	**⇒ r5**
mult	**r3, r5**	**⇒ r6**
sub	**r2, r6**	**⇒ r7**

Extending the Algorithm

More complex cases for **IDENTIFIER**

- What about values in registers?
  - **Already in a register** : return the register name
  - Not in a register : **load it** as before, but record the fact
- What about **parameter values**?
  - Many linkages pass the first several values in registers
  - Call-by-value : just use the offset
  - Call-by-reference : needs an extra indirection
- What about **function calls in expressions**?
  - Generate the calling sequence & load the return value

Extending the Algorithm

- **Adding other operators**
  - Evaluate the operands, then perform the operation
  - Complex operations may turn into **library calls**
  - Handle assignment as an operator
- **Mixed-type expressions**
  - Insert conversions as needed from conversion table
  - Most languages have symmetric & rational conversion tables

Typical Addition Table

+	Integer	Real	Double	Complex
Integer	Integer	Real	Double	Complex
Real	Real	Real	Double	Complex
Double	Double	Double	Double	Complex
Complex	Complex	Complex	Complex	Complex

Handling Assignment

$lhs \leftarrow rhs$

Strategy

- Evaluate *rhs* to a value *(an* ***rvalue****)*
- Evaluate *lhs* to a location *(an* ***lvalue****)*
  - *lvalue* is a register $\Rightarrow$ move rhs
  - *lvalue* is an address $\Rightarrow$ store rhs
- If *rvalue* & *lvalue* have **different types**
  - Evaluate *rvalue* to its "*natural*" type
  - Convert that value to the type of **lvalue*

Evaluation Order

What about evaluation order?

- Can use **commutativity** & **associativity** to improve code
- This problem is truly hard

What about **order of evaluating operands**?

- 1^{st} operand must be preserved while 2^{nd} is evaluated
- Takes an extra register for 2^{nd} operand

Generating Code in the Parser

- Need to generate an initial IR form
  - Chapter 4 talks about AST's & ILOC
  - **AST 생성 후, tree traversal algorithm으로 code 생성**

- The big picture
  - Recursive algorithm **really works bottom-up**
    - Actions on non-leaves occur after children are done
  - Can encode **same basic structure** into **ad-hoc SDT scheme**
    - Identifiers load themselves & stack **virtual register** name
    - Operators emit appropriate code & stack resulting VR name
    - Assignment requires evaluation to an lvalue or an rvalue

Ad-hoc SDT versus a Recursive Treewalk

```
expr(node) {
  int result, t1, t2;
  switch (type(node)) {
      case ×,÷,+,− :
          t1← expr(left child(node));
          t2← expr(right child(node));
          result ← NextRegister( );
          emit(op(node), t1, t2, result);
          break;
      case IDENTIFIER:
          t1← base(node);
          t2← offset(node);
          result ← NextRegister( );
          emit(loadAO, t1, t2, result);
          break;
      case NUMBER:
          result ← NextRegister( );
          emit(loadI, val(node), none, result);
          break;
   }
   return result;
}
```

```
Goal :      Expr  { $$ = $1; } ;
Expr:       Expr PLUS Term
      { t = NextRegister();
       emit(add,$1,$3,t); $$ = t; }
      |     Expr MINUS Term  {…}
      |     Term { $$ = $1; } ;
Term:       Term TIMES Factor
      { t = NextRegister();
       emit(mult,$1,$3,t); $$ = t; };
      |     Term DIVIDES Factor {…}
      |     Factor { $$ = $1; };
Factor:     NUMBER
      { t = NextRegister();
       emit(loadI,val($1),none, t );
       $$ = t; }
      |     ID
      { t1 = base($1);
        t2 = offset($1);
        t = NextRegister();
       emit(loadAO,t1,t2,t);
       $$ =  t; }
```

7.4 Boolean and Relational Operators

Representation for Boolean Values

- **numerical encoding**
  - true, false 각각을 숫자로 표현
  - 계산은 arithmetic & logical operation 사용
- **positional encoding**
  - executable code 상의 위치로 true / false 판별
  - 계산은 comparison & conditional branch 사용

- 각각은 case-by-case로 적합한 경우가 있음

Numerical Encoding

- false = 0
- true = 1 or (NOT 0)

- hw에서 true / false 지원하는 경우: 비교적 간단

- x < y

  cmp_LT rx, ry $\Rightarrow$ r1

- hw에서 condition code만 지원하는 경우:

- x < y

```
    comp rx, ry ⇒ cc1
    cbr_LT cc1 ⇒ L1, L2
L1: loadI true ⇒ r2
    jumpI ⇒ L3
L2: loadI false ⇒ r2
    jumpI ⇒ L3
L3: nop
```

Positional Encoding

- **true/false**를 저장할 필요가 없는 경우
- 특히, if 문 등에서 장점

- x < y (numerical encoding과 동일)

comp r_x, r_y $\Rightarrow$ cc_1
cbr_LT cc_1 $\Rightarrow$ L_1, L_2
L_1: loadI true $\Rightarrow$ r_2
jumpI $\Rightarrow$ L_3
L_2: loadI false $\Rightarrow$ r_2
jumpI $\Rightarrow$ L_3
L_3: nop

- if (x < y)
  then stmt1
  else stmt2

comp r_x, r_y $\Rightarrow$ cc_1
cbr_LT cc_1 $\Rightarrow$ L_1, L_2
L_1: *code for stmt*$_1$
jumpI $\Rightarrow$ L_3
L_2: *code for stmt*$_2$
jumpI $\Rightarrow$ L_3
L_3: nop

Hardware Support for Relational Op

- if (x < y)
  then a ← c + d
  else a ← e + f
- straight condition codes

```
     comp r_x, r_y ⇒ cc_1
     cbr_LT  cc_1 ⇒ L_1, L_2
L_1: add  r_c, r_d ⇒ r_a
     jumpI ⇒ L_3
L_2: add  r_e, r_f ⇒ r_a
     jumpI ⇒ L_3
L_3: nop
```

- conditional move

```
     comp r_x, r_y ⇒ cc_1
     add  r_c, r_d ⇒ r_1
     add  r_e, r_f ⇒ r_2
     i2i_LT cc_1, r_1, r_2 ⇒ r_a
```

- Boolean compare

```
     cmp_LT r_x, r_y ⇒ r_1
     cbr  r_1 ⇒ L_1, L_2
L_1: add  r_c, r_d ⇒ r_a
     jumpI ⇒ L_3
L_2: add  r_e, r_f ⇒ r_a
     jumpI ⇒ L_3
L_3: nop
```

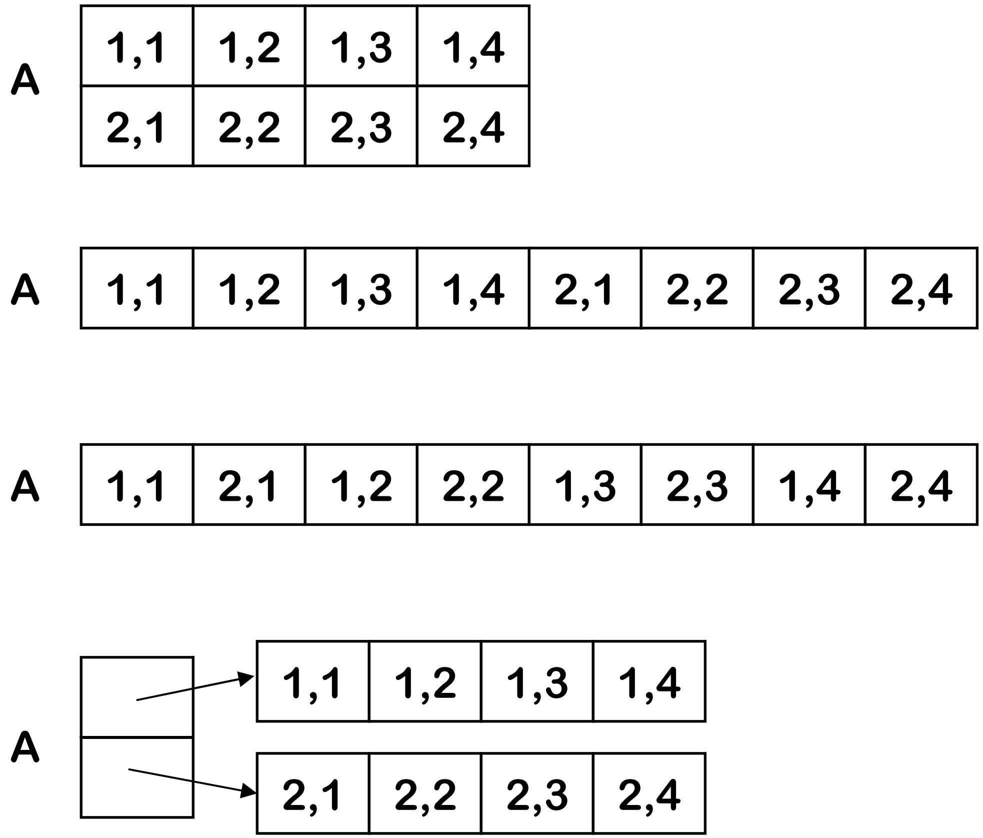
- predicated execution

```
            cmp_LT r_x, r_y ⇒ r_1
            not  r_1 ⇒ r_2
(r_1)?      add  r_c, r_d ⇒ r_a
(r_2)?      add  r_e, r_f ⇒ r_a
```

7.5 Storing and Accessing Arrays

Representing Arrays

- concept
- **row-major order**
  - C, C++, …
- column-major order
  - ForTran
- **indirection vectors**
  - Java



Referencing an Array Element

- 1-D array case : A[*low*..*high*]
  - A[*low*], A[*low*+1], …, A[*high*–1], A[*high*]
- to get A[*i*]

  base(A) + (*i* – *low*) * sizeof(A[*low*])
- to optimize it:
  - let *low* = 0
    - "–" 연산을 하지 않는다.
  - sizeof(A[*low*]) 를 2의 제곱이 되게 한다.
    - shift 연산으로 가능 → "*" 를 하지 않는다.

Referencing an Array Element

- What about A[i_1,i_2] ?
- Row-major order, 2D

  @A + ((i_1 – low_1) * ($high_2$ – low_2 + 1) + i_2 – low_2) * sizeof(A[1])
- Column-major order, 2D

  @A + ((i_2 – low_2) * ($high_1$ – low_1 + 1) + i_1 – low_1) * sizeof(A[1])
- Indirection vectors, 2D

  *(A[i_1])[i_2]
  - where A[i_1] is, itself, a 1D array reference

Optimizing Address Calculation

In row-major order

$$@A + (i - low_1)(high_2 - low_2 + 1) * w + (j - low_2) * w$$

Which can be factored into

$$@A + i * (high_2 - low_2 + 1) * w + j * \mathrm{w}$$
$$- (low_1 \text{ x } (high_2 - low_2 + 1) * w) + (low_2 * w)$$

If low_i, $high_i$, and w are known, the last term is a constant

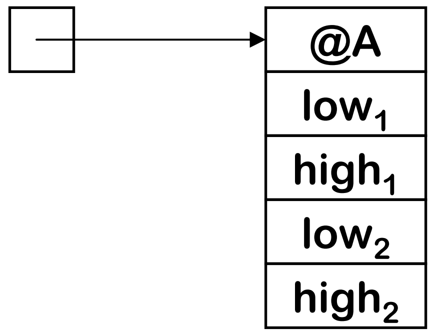
Define $@A_0 = @A - (low_1 * (high_2 - low_2 + 1) * w + low_2 * w$

And $len_2 = (high_2 - low_2 + 1)$

Then, the address expression becomes

$$\boldsymbol{@A_0 + (i * len_2 + j) * w}$$

Array Reference



- What about arrays as actual parameters?
- call-by-reference case
  - Need **dimension information**
    → build a **dope vector**
  - Store the values in the calling sequence
  - Pass the address of the dope vector in the parameter slot
  - Generate complete address polynomial at each reference
- call-by-value case
  - Most call-by-value languages pass arrays by reference
  - This is a language design issue

7.6 Character Strings

String Representation

- 근본적으로, **1-D array of characters**
- C-like languages: **(array + '\0')** style
  - 장점: 간단, 적은 memory
  - 단점: string length 계산 등이 어려움

a	b	c	d	e	f	\0

- other languages: **(string length + array)**

length = 6	a	b	c	d	e	f

7.7 Structure References

Structure Representation

- ```
  struct node {
          int value;
          struct node* next;
  };
  struct node NilNode = { 0, (struct node*) 0};
  struct node* NIL = &NilNode;
  ```
- two problems
  - pointer → **anonymous values**
    - **pointer**로 만든 **struct**는 **name**이 없다.
  - **struct layout** : 어떻게 저장할 것인가
  ```

Structure Representation

- struct layout table : single table approach

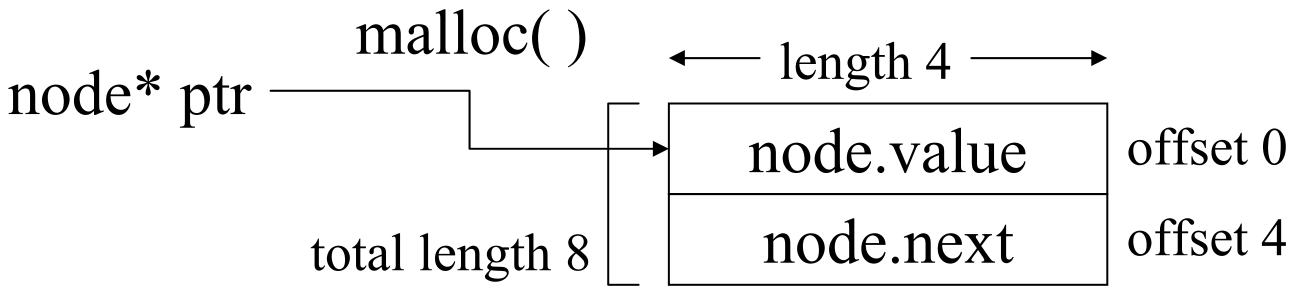
structure layout table

name	length	start
node	8	0
…	…	2

element table

name	length	offset	type	next
node.value	4	0	int	
node.next	4	4	node*	
…	…	…	…	…

NULL



7.8 Control-Flow Constructs

If-Then-Else 구현

- Boolean 구현에서 이미 다루었음

- x < y

  comp $r_x, r_y \Rightarrow cc_1$

  cbr_LT $cc_1 \Rightarrow L_1, L_2$

L_1: loadI true $\Rightarrow r_2$

jumpI $\Rightarrow L_3$

L_2: loadI false $\Rightarrow r_2$

jumpI $\Rightarrow L_3$

L_3: nop

- if (x < y)
  then stmt1
  else stmt2

  comp $r_x, r_y \Rightarrow cc_1$

  cbr_LT $cc_1 \Rightarrow L_1, L_2$

L_1: *code for stmt*$_1$

jumpI $\Rightarrow L_3$

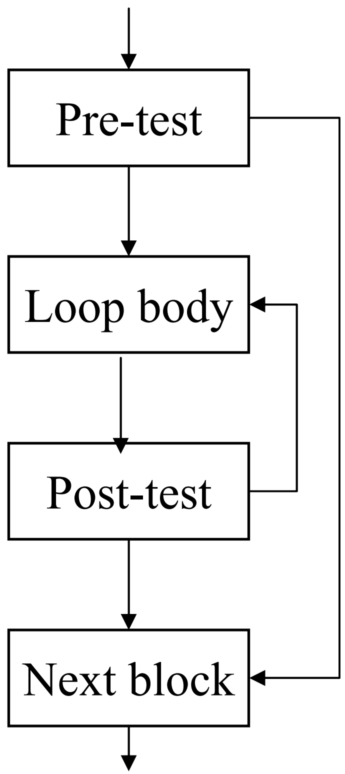
L_2: *code for stmt*$_2$

jumpI $\Rightarrow L_3$

L_3: nop

Loop 구현

- basic model
  - **pretest** (if needed)
    - evaluate condition before loop
    - branch to the next block, if needed
  - **loop body**
  - **posttest** (if needed)
    - evaluate condition after loop
    - branch back to the top, if needed



- **while**, **for**, **do** & **until** all fit this basic model

Loop 구현의 예: for-loop

- for (i = 1; i <= 100; i++) {
  body
  }
  next statement

  loadI 1 $\Rightarrow$ r_i — Initialization
  loadI 100 $\Rightarrow$ r_1
  cmp_GE r_i, r_1 $\Rightarrow$ r_2 — Pre-test
  cbr r_2 $\Rightarrow$ L_2, L_1
L_1: *body*
  addI r_i, 1 $\Rightarrow$ r_i
  cmp_LE r_i, r_1 $\Rightarrow$ r_3 — Post-test
  cbr r_5 $\Rightarrow$ L_1, L_2
L_2: *next statement*

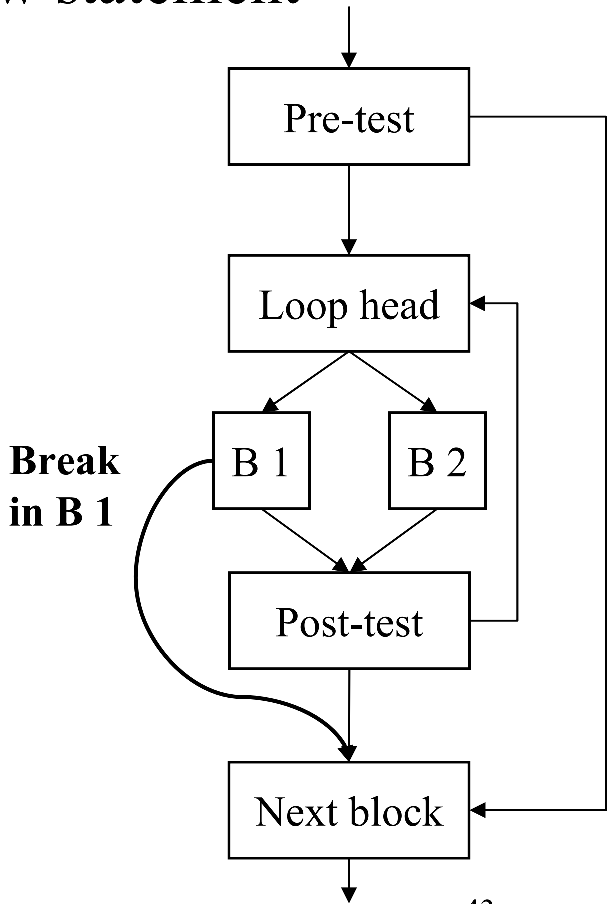
Switch-Case 구현

- switch (*expression*) {
  case $expr_1$: $stmt_1$; break;
  case $expr_2$: $stmt_2$; break;
  …
  }
  next statement
- step 1: evaluate the controlling expression
- step 2: branch to the selected case
  - **how ?** → linear search, binary search, table lookup, hashing, …
- step 3: execute the code for that case
- step 4: branch to the statement after the case

break 구현

- Many modern programming languages include a break
  - **Exits** from the innermost control-flow statement
  - Out of the innermost loop
  - Out of a case statement

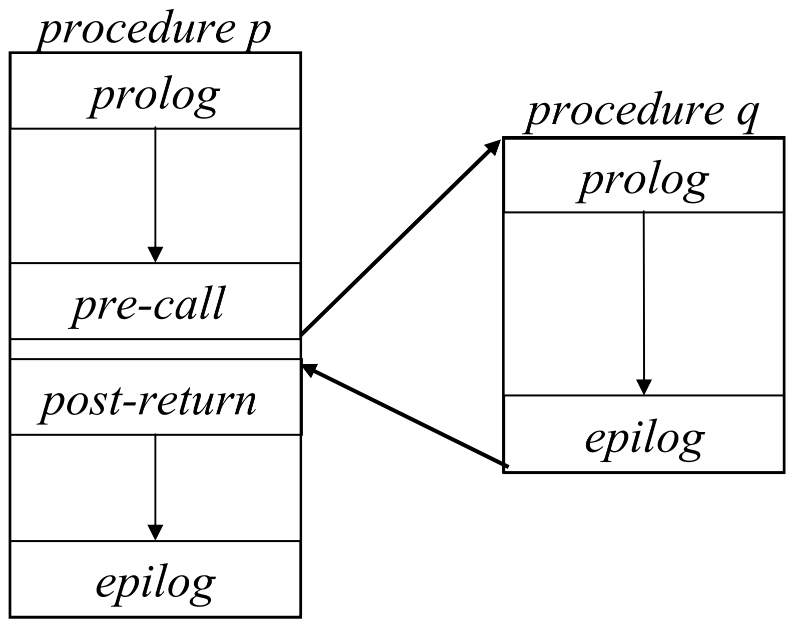
- **Translates into a jump**
  - Targets statement outside control-flow construct
  - Creates multiple-exit construct
  - Skip in loop goes to next iteration



7.9 Procedure Calls

Standard Linkage

- 전체 구조는 이미 chap 6에서 했음
- compiler 관점에서는
  - **pre-call, post-return 부분은 되도록 간단하게**
  - call 하는 부분마다 삽입되므로, 전체 code 양이 늘어날 수 있음



- **lead procedure**
  - 자신은 call 을 하지 않는 경우
    - 특히, inline인 경우
  - code를 더 간략하게 줄일 수 있음

Implementing Procedure Calls

- If *p* calls *q* :

- In the code for *p*, compiler emits **pre-call sequence**
  - evaluates each parameter & stores it appropriately
  - branches to the entry of *q*
- In the code for *p*, compiler emits **post-return sequence**
  - copy return value into appropriate location
  - free *q*'s AR, if needed
  - resume *p*'s execution

Implementing Procedure Calls

- In the prolog, ***q* must**
  - set up its execution environment
  - allocate space for its (AR &) local variables & initialize them
  - establish addressability for static data area(s)
- In the epilog, ***q* must**
  - store return value
  - restore the return address (if saved)
  - begin restoring *p*'s environment
  - load return address and branch to it

Implementing Procedure Calls

If *p* calls *q*, one of them must:

- **preserve register values**
  - caller-saves registers stored/restored by *p* in *p*'s AR
  - callee-saves registers stored/restored by *q* in *q*'s AR
- **allocate the AR**
  - heap allocation: callee allocates its own AR
  - stack allocation:
    caller & callee cooperate to allocate AR

7.10 Implementing OOL

OOL에서의 구현

- chap 6에서 기본 아이디어는 나왔음
- 구현 상의 주의점
  - 모든 **method call**을 **indirect**로 구현해야
  - 즉, class에 저장된 address를 가져와서 call