Chap 6. The Procedure Abstraction

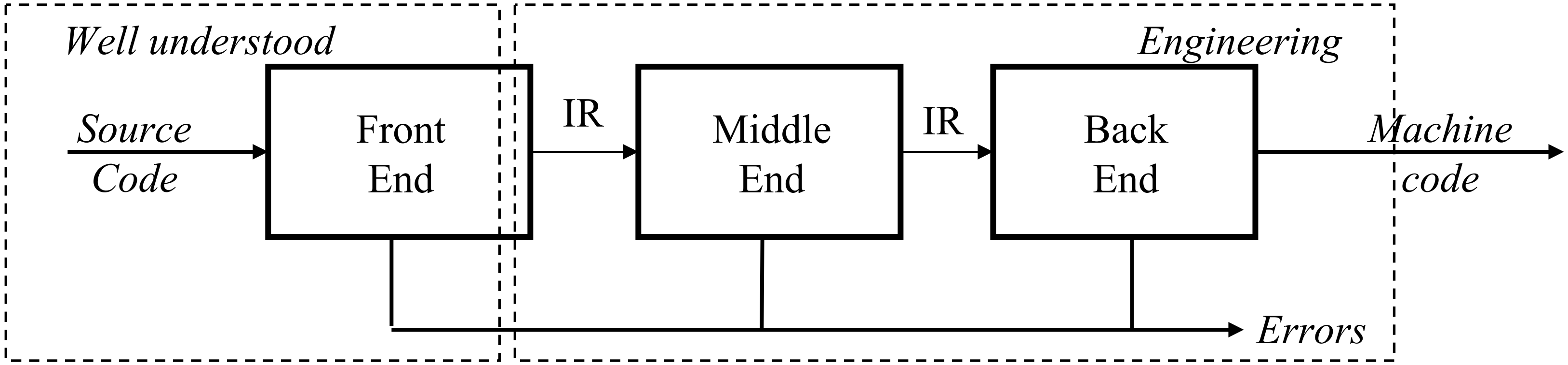
COMP321 컴파일러

2007년 가을학기

경북대학교 전자전기컴퓨터학부

6.1 Introduction

Where are we ?



The latter half of a compiler contains ***more open problems****,* *more challenges, and more gray areas than the front half.*

- This is "compilation," as opposed to "parsing" or "translation"
- Implementing promised behavior
  - What defines the meaning of the program
- Managing target machine resources

Procedure Abstraction

- **procedure**
  - 대부분의 PL에서 가장 중요한 abstraction
  - 별도의 name space를 가지고, 별도의 환경을 제공
  - function = procedure with return value

- **separate compilation**
  - procedure는 독립적 요소
  - compile 시, 분리해서 compile 가능 !

Procedure: Three Abstractions

- **Control Abstraction**
  - Well defined entries & exits
  - caller, callee 간의 control 전환 방법 필요
  - calling convention : parameter passing 방법 필요
- **Clean Name Space**
  - 별도의, new protected name space 생성
  - non-local name이나, 이전 name space보다 우선
- **External Interface**
  - Access is by procedure name & parameters
  - Clear protection for both caller & callee
- Procedures permit a critical separation of concerns

Procedure: Realist's View

- **the key to building large systems**
  - algorithm을 작은 단위로 구현 가능
  - 완전히 abstract한 별도의 operation
- **separate compile** 가능
  - 사람이 느끼는 compile time을 줄이는 효과
  - co-work 이 가능해진다.
- **linkage convention** 제공
  - 서로 다른 PL 에서도 calling 가능

Run Time vs. Compile Time

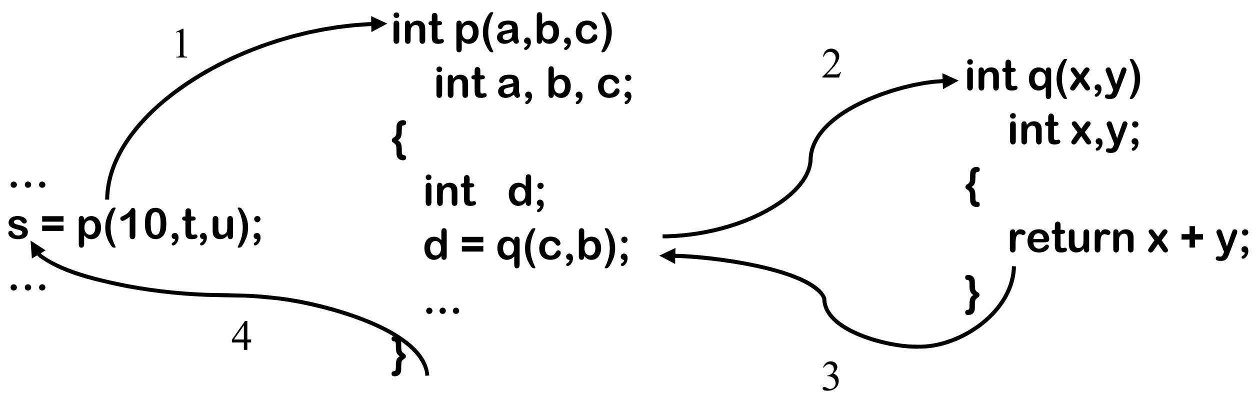
- procedure를 어떻게 구현할 것인가?

- **linkage** : run-time에 수행됨
  – 실제 어느 procedure를 부를 지는 run-time에 결정
- **code for linkage** : compile-time에 생성
  – procedure call을 수행하는 code는 compile 때 생성
- **design for linkage** : 미리 결정
  – 어떻게 서로 call / return 할 것이지는 미리 design
  – PL 설계 시나, compiler 제작 시에 미리 결정

6.2 Control Abstraction

Procedure as a Control Abstraction

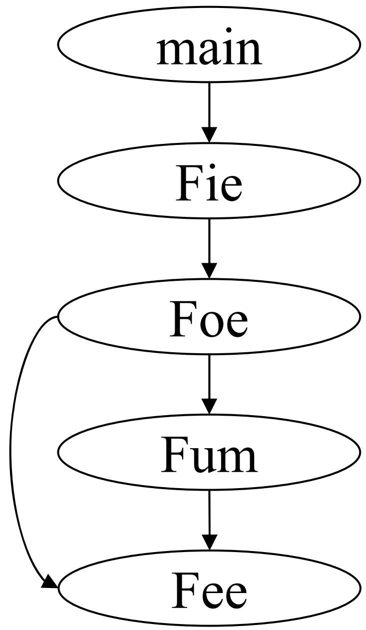
- Procedures have well-defined control-flow
- The **Algol-like procedure call**
  - Invoked at a call site, with some set of actual parameters
  - Control returns to call site, immediately after invocation
- Most PL allows recursions.



Call Graph

```
program main(input, output);
   procedure Fee;
      begin { Fee }
         …
      end;
   procedure Fie;
      procedure Foe;
         procedure Fum;
            begin { Fum }
               Fee
            end;
         begin { Foe }
            Fee;
            Fum
         end;
      begin { Fie }
         Foe
      end;
   begin { main }
      Fie
   end;
```

- call graph
  - the set of potential calls among the procedures
  - procedure 구현의 기본적 분석 자료



Activation of an Instance

- procedure는 dynamic하게 invoke 된다
  - **instance** : procedure의 invoke 된 상태
    - 일반적인 경우 : one instance per one procedure
    - recursion 상황 : multiple instance for a procedure
- **activation**
  - distinct instance (실제로는 혼용해서 사용)
  - call : activation 을 만든다
  - return : activation이 사라짐
  - 구현?   use stack !   → recursion도 가능

Recursion

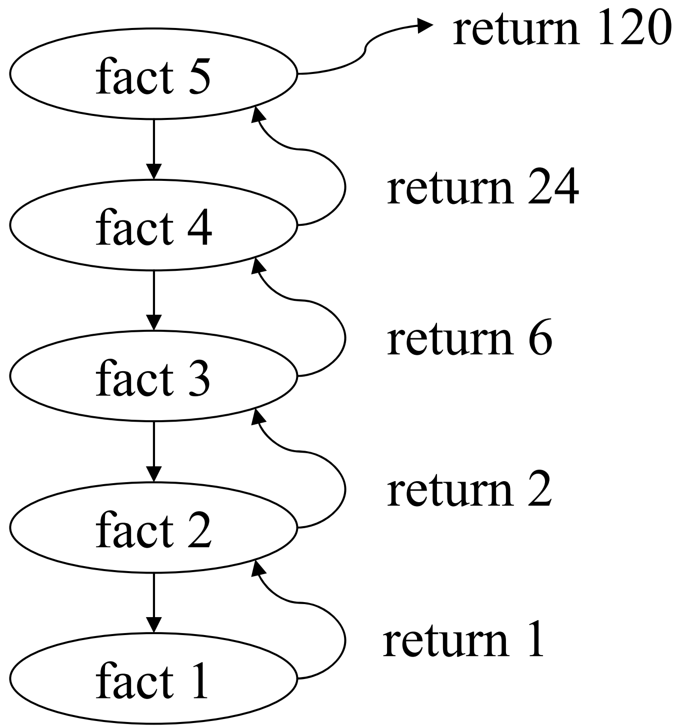
- lisp program

```
(define (fact k)
    (cond
        [ (<= k 1) 1]
        [else (* (fact (sub1 k)) k)]
    ))
```

- C program

```
int fact(int k) {
    if (k <= 1) return 1;
    else return fact(k – 1) * k;
}
```

- activation 상황



Procedure as a Control Abstraction

Implementing procedures with this behavior

- “**return address**”를 save & later load 하는 기능
- actual parameter → formal parameter로의 **mapping** ($c \rightarrow x$, $b \rightarrow y$)
- must create storage for **local variables** (&, maybe, parameters)
  - *p* needs space for *d* (also, maybe, *a, b*, & *c*)
  - where does this space go in recursive invocations?

```
…
s = p(10,t,u);
…
```

```
int p(a,b,c)
   int a, b, c;
{
  int   d;
  d = q(c,b);
  …
}
```

```
int q(x,y)
   int x,y;
{
   return x + y;
}
```

*Compiler <u>emits</u> code that causes all this to happen **at run time***

Procedure as a Control Abstraction

Implementing procedures with this behavior

- Must preserve ***p*'s state** while *q* executes
  - recursion causes the real problem here
- *Strategy*: Create unique location for each procedure activation
  - use a "**stack**" of memory blocks to hold local storage and return addresses

```
…
s = p(10,t,u);
…
```

```
int p(a,b,c)
  int a, b, c;
{
  int   d;
  d = q(c,b);
  …
}
```

```
int q(x,y)
  int x,y;
{
  return x + y;
}
```

Compiler <u>emits</u> code that causes all this to happen ***at run time***

6.3 Name Spaces

Name Space

- name space : 독립적으로 **name**을 관리하는 공간
  - name : variable name, function name, …
  - scope = each name space
- Algol-like languages
  - including C, Pascal, C++, Java, …
  - nested scope
    - block in block : C, C++, Java
    - procedure in procedure: Pascal
    - **class in class** : C++, Java

Procedure as a Name Space

Each procedure creates its own name space

- Any name (almost) can be declared locally
- Local names obscure identical non-local names
- Local names cannot be seen outside the procedure
- We call this set of rules & conventions "**lexical scoping**"

Example: C programming language

- one global name space
- multiple nested block scopes

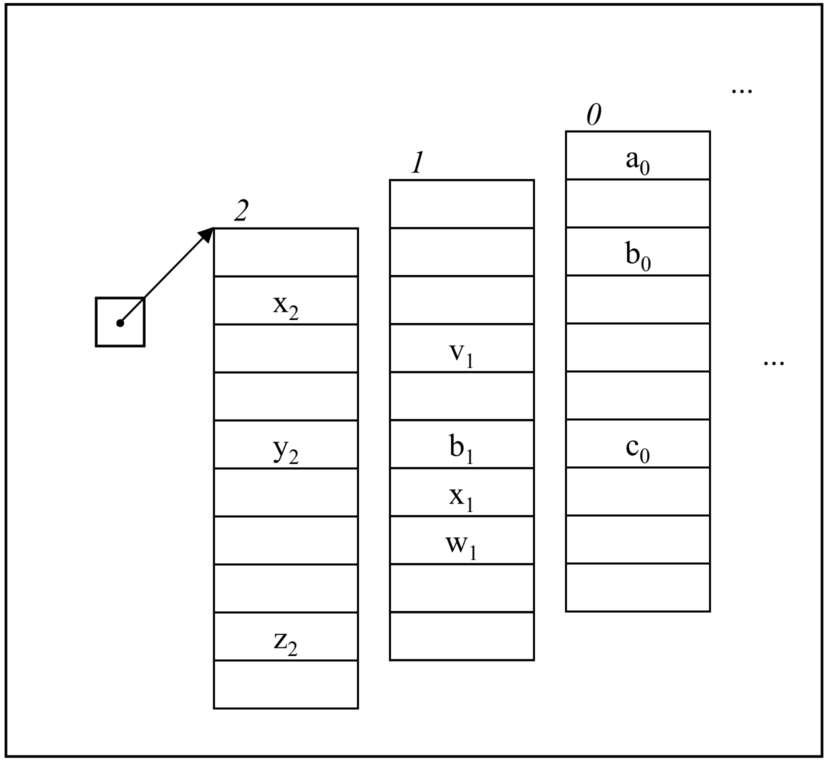
Lexical Scoping

- Why introduce lexical scoping?
  - Provides a compile-time mechanism for binding “free” variables
  - Simplifies rules for naming & resolves conflicts
- How can the compiler keep track of all those names?
  - The Problem
    - At point *p*, **which declaration of *x* is current?**
    - At run-time, **where is *x* found?**
    - As parser goes in & out of scopes, how does it delete *x*?
  - The Answer: **lexically scoped symbol tables**

Lexically Scoped Symbol Table

```
B0: {
        int a0, b0, c0
B1:     {
            int v1, b1, x1, w1
B2:         {
                int x2, y2, z2
                ....
            }
B3:         {
                int x3, a3, v3
                ...
            }
            ...
        }
        ...
    }
```

- linked list 형태의 symbol table 관리 : see Section 5.7
  - 주의: **compile time 임 !**



Run-time 관리는 ?

variable의 life-time 분석

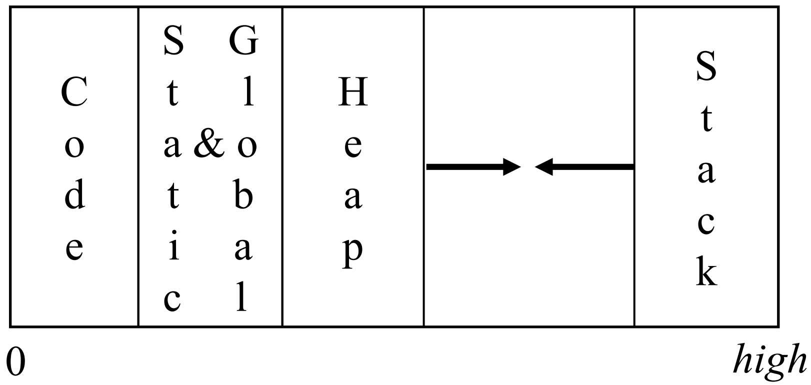
- life-time = 해당 variable이 memory에 유지되는 time
- **automatic** (or local) :
  해당 procedure와 같은 life-time
- **static in a procedure** : never expire !
- **static in a file** : never expire !
- **global** : never expire !
- malloc( ), new : **dynamic** memory allocation …

run-time 구현은 어떻게?

- one large, global memory area
- multiple memory area for each procedure

Placing Run-time Data Structures

- classical data organization



Single Logical Address Space

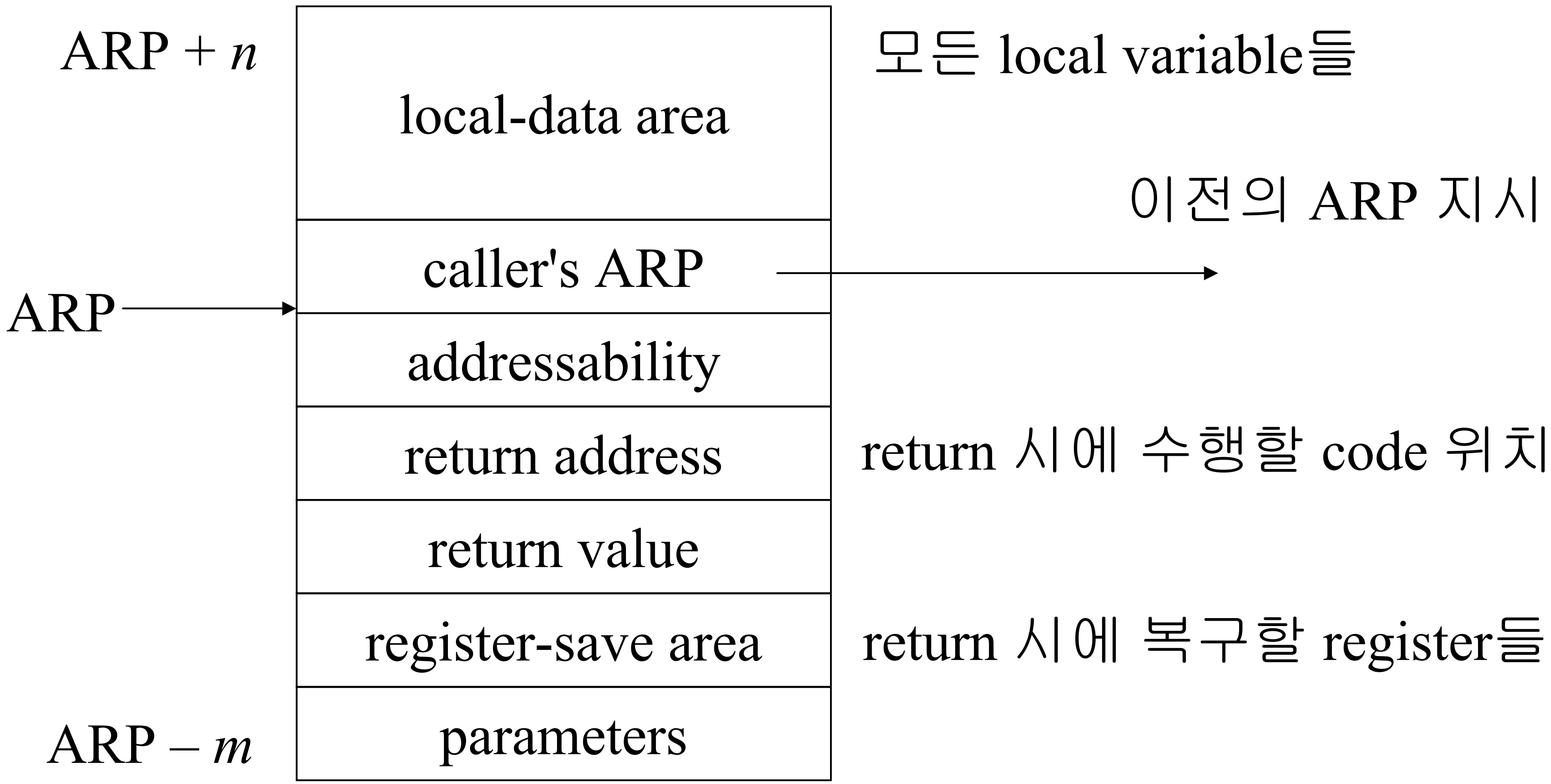
heap: malloc(), new를 처리
stack : local variable 저장

- code, static & global data have known size
- use symbolic labels in the code
- heap & stack both grow & shrink over time

Activation Record Model

- **activation record** (AR)
  - procedure의 activation 마다 별도의 공간 할당
  - compile time의 **symbol table 과 밀접한 관계**
  - run-time에는 **stack에 올라감**
- **ARP : activation record pointer**
  - 지금 이 순간 activate된 AR을 pointing
  - AR에는 다음 AR을 가리키는 pointer 설정
  - 일종의 linked list 구조 !
  - ARP는 자주 사용되므로, 보통 register에 저장

AR의 일반적 구조


ARP + *n*
local-data area
모든 local variable들
이전의 ARP 지시
caller's ARP
ARP
addressability
return address
return 시에 수행할 code 위치
return value
register-save area
return 시에 복구할 register들
ARP – *m*
parameters

ARP의 실제 구현 예

```
int func(int a, int b) {
    int c;
    int d;
    c = a * b;
    d = a / b;
    a = c + d;
    return a;
}

int main(void) {
    int k;
HERE:   k = func(27, 3);
    …
}
```

ARP + 8	**local-data: int c**
ARP + 4	**local-data: int d**
ARP + 0	caller's ARP: main's ARP →
ARP – 4	addressability
ARP – 8	return address: HERE
ARP – 12	return value: int type
ARP – 16 … ARP – 76	register-save area (totally 16 * 4 bytes)
ARP – 80	**parameter: a**
ARP – 84	**parameter: b**

ARP의 실제 구현 예: code 생성

```
int func(int a, int b) {
    int c;
    int d;
    c = a * b;
*(ARP + 8) ← *(ARP – 80) * *(ARP – 84)
    d = a / b;
*(ARP + 4) ← *(ARP – 80) / *(ARP – 84)
    a = c + d;
*(ARP – 80) ← *(ARP + 8) + *(ARP + 4)
    return a;
*(ARP – 12) ← *(ARP – 80)
…
stack pop for the AR
jump (ARP – 8)
}
```

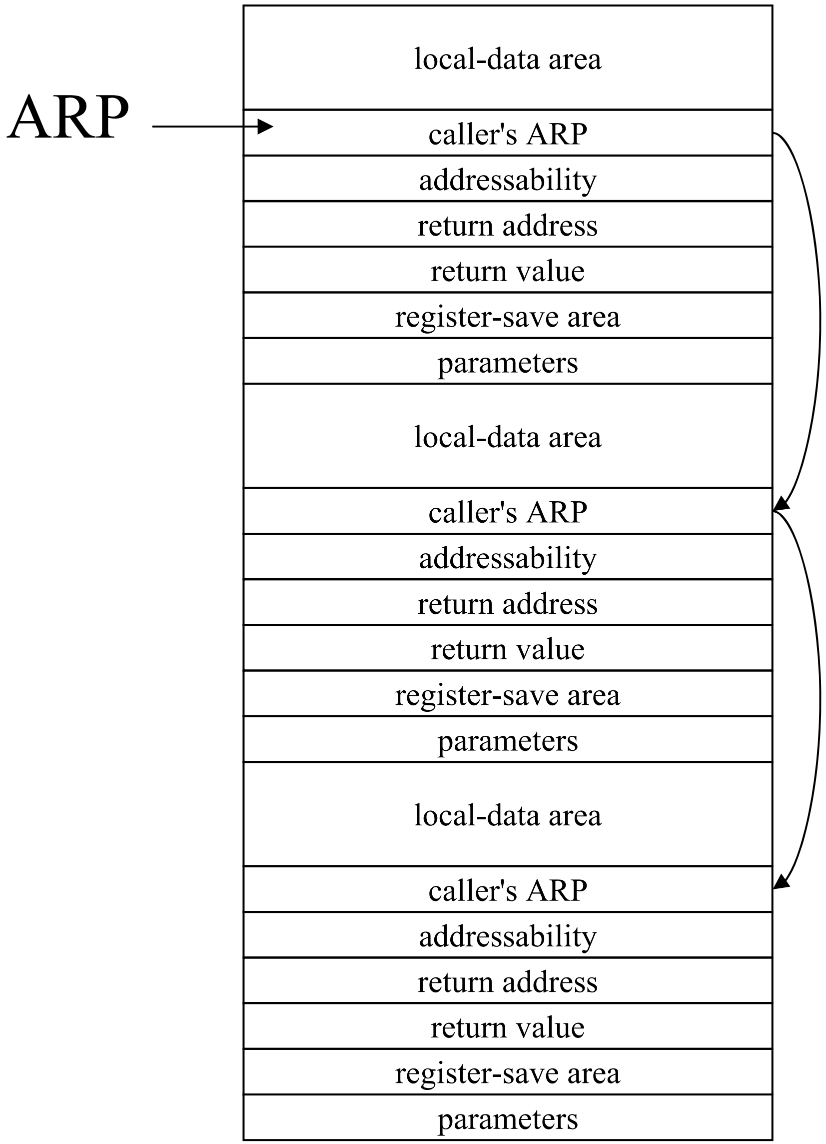
ARP + 8	**local-data: int c**
ARP + 4	**local-data: int d**
ARP + 0	caller's ARP: main's ARP
ARP – 4	addressability
ARP – 8	return address: HERE
ARP – 12	return value: int type
ARP – 16 … ARP – 76	register-save area (totally 16 * 4 bytes)
ARP – 80	**parameter: a**
ARP – 84	**parameter: b**

Translating Local Names

- Name is translated into a *static coordinate*
  - < *level, offset* > pair
  - *"level"* is lexical nesting level of the procedure
    - 즉, 해당 symbol table 또는 AR을 가리킴
  - *"offset"* is *unique* within that scope
    - ARP 와의 거리
    - symbol table에 저장 → code 생성에 사용
    - run-time : 주소만 이용 → 이제 name은 불필요

ARP stack의 구현

- ARP는 stack에 차례 대로 쌓임
- 이전 block으로 연결
  - caller's ARP를 따라감.



AR의 구현 방법

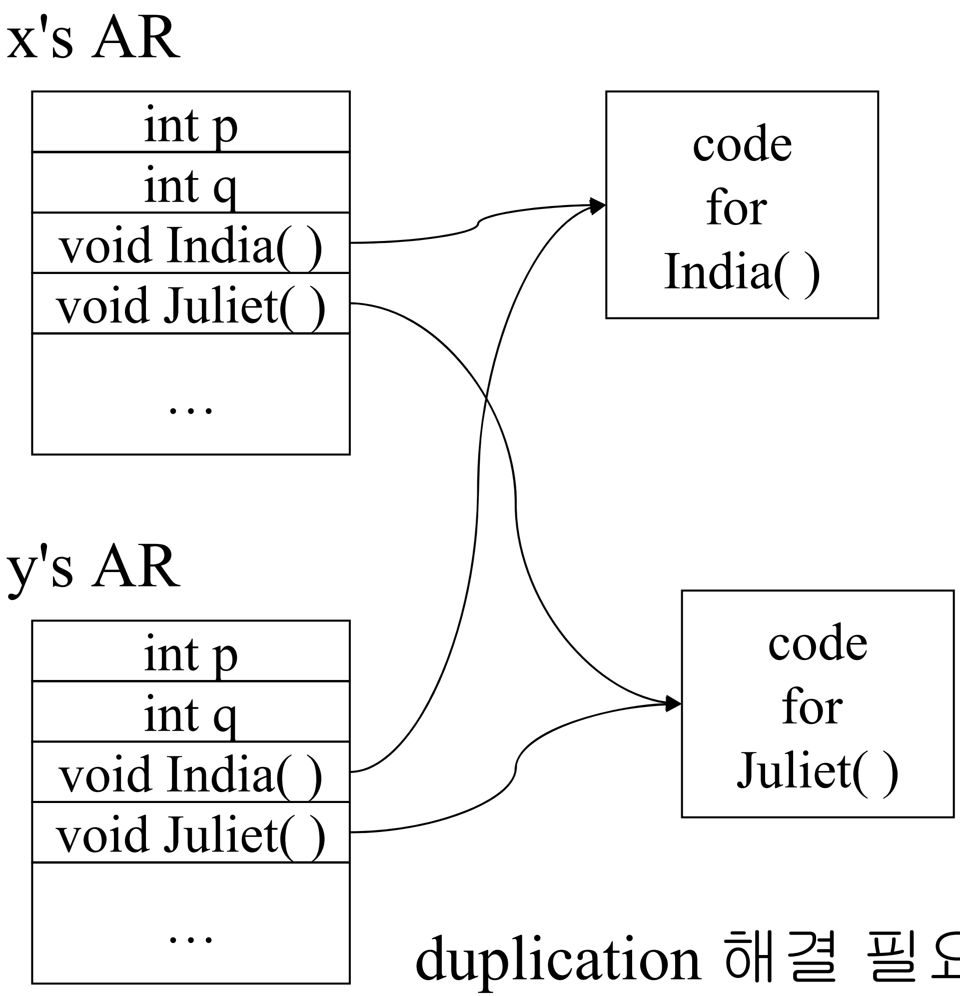
- **stack** 으로 구현
  – Algol-like PL에서 가장 일반적인 방법
- **heap** 으로 구현
  – 일부 PL 에서는 return 시에, 꼭 순서대로 pop 되는 것은 아니다. (예: ML)
  – heap 으로 AR을 구현해야 함
- **static allocation** 으로 구현
  – recursion 이 없는 PL 에서는 가능 (예: ForTran)
  – 속도가 조금 빨라짐

What about Object-Oriented Languages?

- What is an OOL?
  - a PL that supports "object-oriented programming"
- How does an OOL differ from an ALL?
  - ALL = ALGOL-Like Language
  - **data-centric name scopes** for values & functions
  - dynamic resolution of names to their implementations
    - **class hierarchy** 에 따라, 처리해야
- How do we compile OOLs ?
  - need to define what we mean by an OOL

OOL: Method Pointers are Required

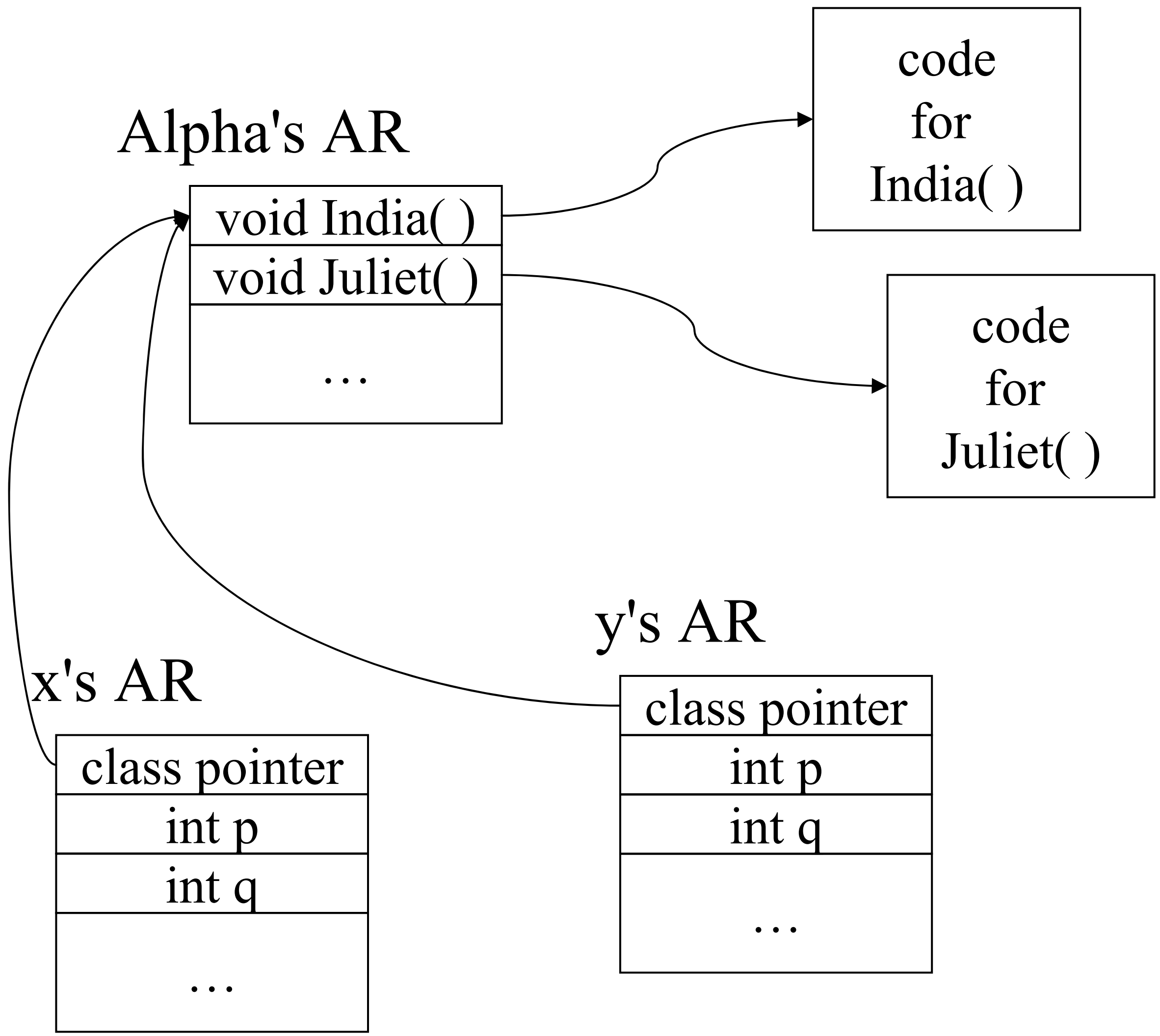
```
class Alpha {
    int  p, q;
    void  India(…);
    void  Juliet(…);
};

Alpha  x;
Alpha  y;
…
x.India(…);
```



duplication 해결 필요 !

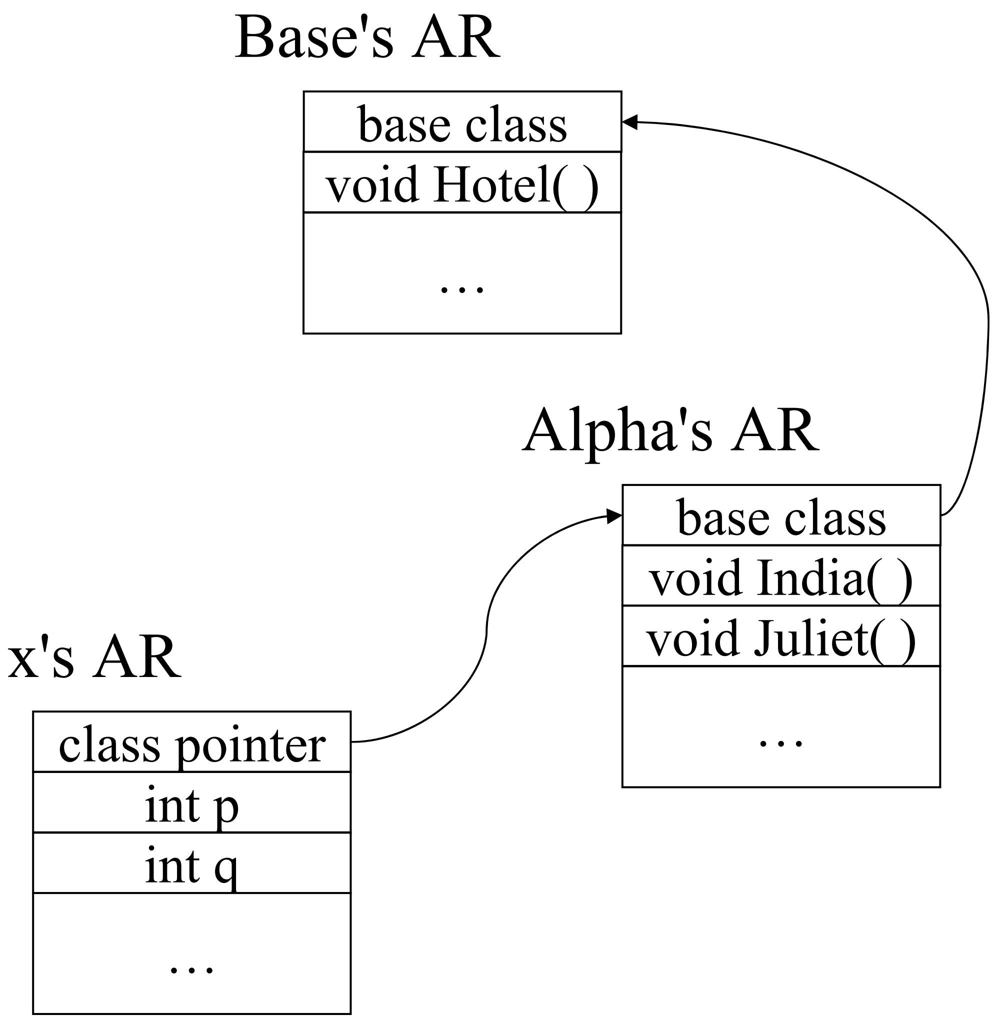
OOL: Class and Instance

```
class Alpha {
    int  p, q;
    void  India(…);
    void  Juliet(…);
};

Alpha  x;
Alpha  y;
…
x.India(…);
```



OOL: Inheritance의 구현

```
class Base {
    …
    void Hotel(…);
};
class Alpha : public Base {
    int  p, q;
    void  India(…);
    void  Juliet(…);
};

Alpha  x;
…
x.Hotel(…);
```



6.4 Communicating Values Between Procedures

Parameter Passing

- actual parameter → formal parameter
  - How to binding them ?
  - How to update the actual parameter when return ?

- **call by value**
- **call by value-result**
- **call by reference**
- **call by name**

Call by Value, Call by Value-Result

- step 1: calculate the value of actual parameter
- step 2: make local variable on the callee's AR
- step 3: **copy the value** to the callee's AR
  - 이제, formal parameter는 단순한 local variable
  - C / C++의 기본적 방법

- call by value-result
  - At return time,
  - (reverse) copy the local variable value to the actual parameter

Call by Reference

- passes **a pointer** to actual parameter
  - Requires slot in the AR (for address of parameter)
  - pointer 를 이용해서, 직접 값을 바꾼다.
  - C / C++에서 pointer 를 사용할 때의 방법
  - ForTran에서는 유일한 binding 방법

- 장점: array, structure passing에서 유리
  - C++ : const 선언 시, call by value 이더라도, 내부적으로는 call by reference로 구현

6.5 Establishing Addressability

Establishing Addressability

Must create **base addresses**

- Global & static variables
  - Construct **a label** by mangling names (*i.e.,* $$fee)
- Local variables
  - Convert to static data coordinate and use **ARP + offset**
- Local variables of other procedures
  - Convert to static coordinates
  - Find appropriate **ARP**
  - Use that **ARP + offset**

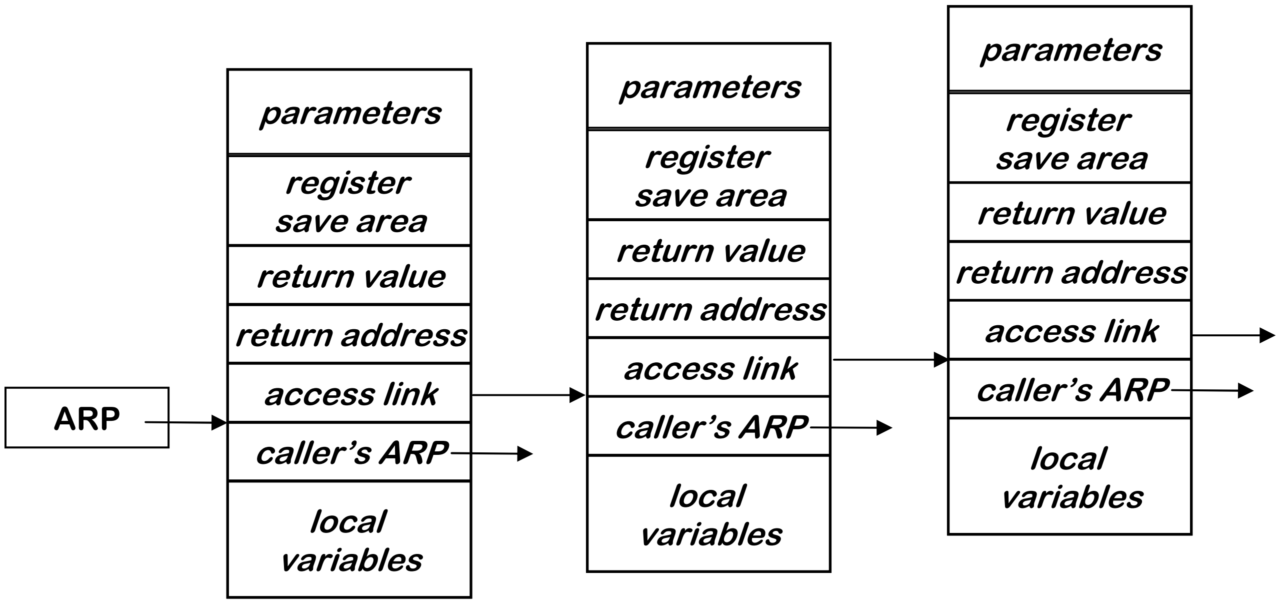
Must find the right AR

Need links to nameable ARs

Establishing Addressability

Using **access links**

- Each AR has a pointer to AR of lexical ancestor
- Lexical ancestor need not be the caller

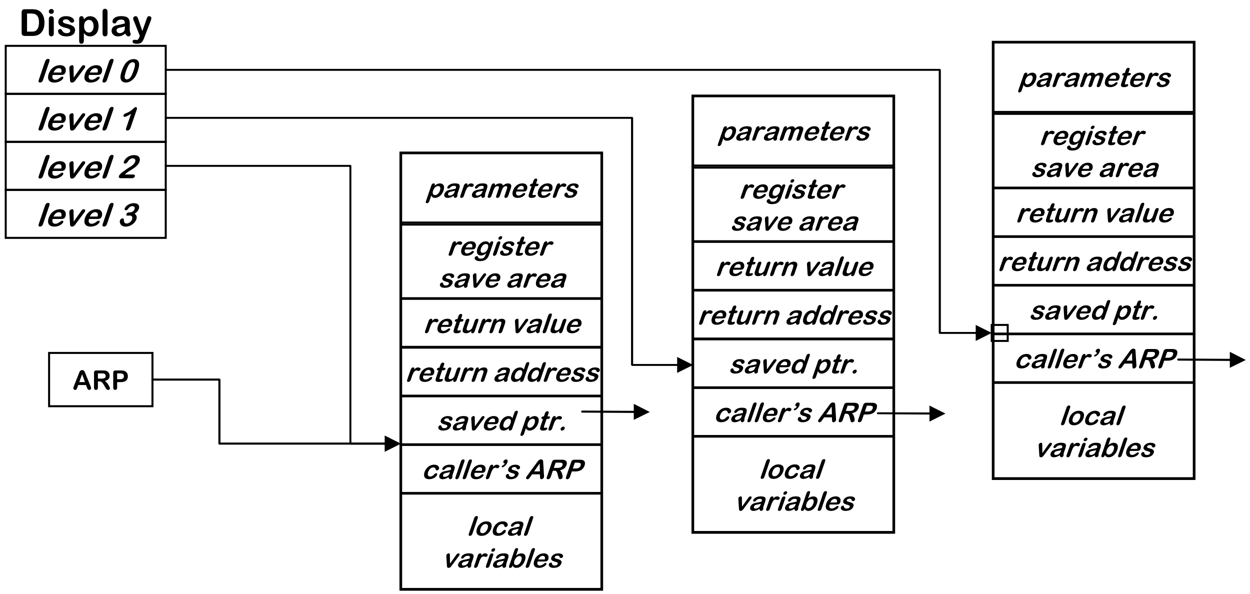


- Reference to <*p*,16> runs up access link chain to *p*
- Cost of access is proportional to lexical distance

Establishing Addressability

Using a **display**

- Global array of pointer to nameable ARs
- Needed ARP is an array access away



- Reference to <*p*,16> looks up *p*'s ARP in display & adds 16
- Cost of access is constant (ARP + offset)

Establishing Addressability

- Access links versus Display
  - Each adds some overhead to each call

- Access links costs vary with level of reference
  - Overhead only incurred on references & calls
- Display costs are fixed for all references
  - References & calls must load display address
  - Typically, this requires a register

6.6 Standardized Linkages

Procedure Linkages

How do procedure calls actually work?

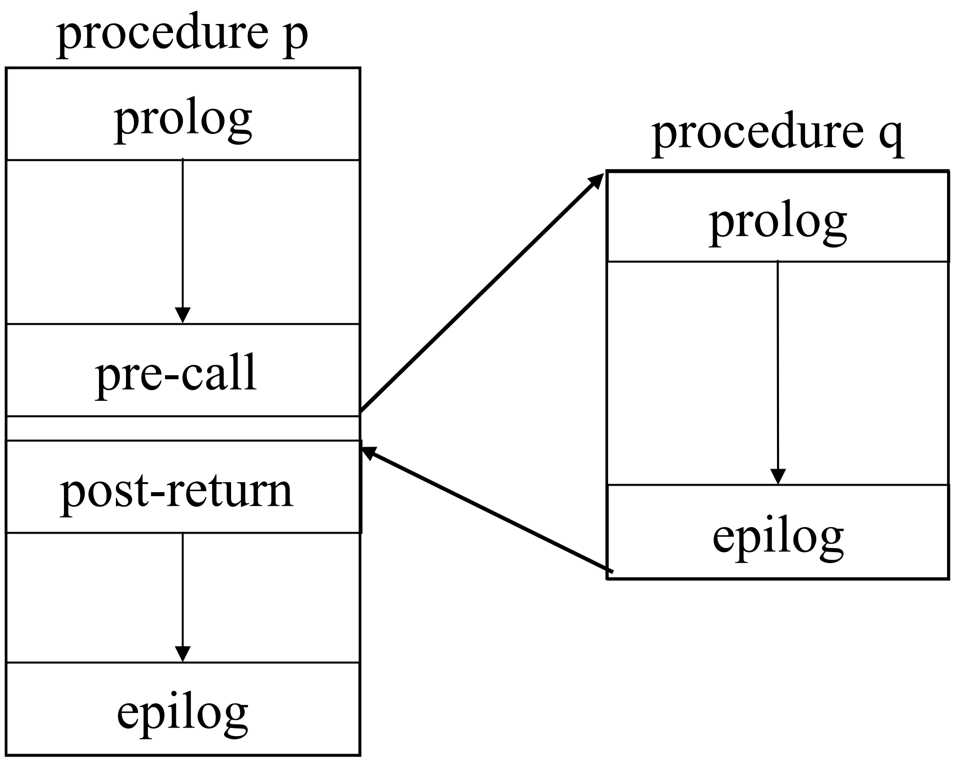
- At compile time, callee may not be available for inspection
  - Different calls may be in different compilation units
  - Compiler may not know system code from user code
  - All calls must use the same protocol

Compiler must use a standard sequence of operations

- Enforces control & data abstractions
- Divides responsibility between caller & callee

Standard Procedure Linkages

Standard procedure linkage



Procedure has

- standard prolog
- standard epilog

Each call involves a

- pre-call sequence
- post-return sequence

These are completely predictable from the call site $\Rightarrow$ depend on the number & type of the actual parameters

Pre-Call Sequence

- Sets up callee's basic AR
- Helps preserve its own environment

The Details

- **Allocate space for the callee's AR**
  - except space for local variables
- Evaluates each parameter & stores value or address
- Saves **return address**, caller's ARP into callee's AR
- Save any **caller-save registers**
- Save into space in caller's AR
- Jump to address of callee's prolog code

Post-Return Sequence

- Finish restoring caller's environment
- Place any value back where it belongs

The Details

- Copy return value from callee's AR, if necessary
- **Free the callee's AR**
- Restore any **caller-save registers**
- Restore any call-by-reference parameters to registers, if needed
- Also copy back call-by-value/result parameters
- Continue execution after the call

Prolog Code

- Finish setting up the callee's environment
- Preserve parts of the caller's environment

The Details

- Preserve any callee-save registers
- **Allocate space for local data**
  - Easiest scenario is to extend the AR
- Find any static data areas referenced in the callee
- Handle any local variable initializations

Epilog Code

- Wind up the business of the callee
- Start restoring the caller’s environment

The Details

- Store return value? No, this happens on the return statement
- **Restore callee-save registers**
- Free space for local data, if necessary (on the heap)
- Load return address from AR
- Restore caller’s ARP
- Jump to the return address

Back to Activation Records

If activation records are stored **on the stack**

- Easy to extend — simply bump top of stack pointer
- Caller & callee share responsibility
  - Caller can push parameters, space for registers, return value slot, return address, addressability info, & its own **ARP**
  - Callee can push space for local variables (fixed & variable size)

If activation records are stored on the heap

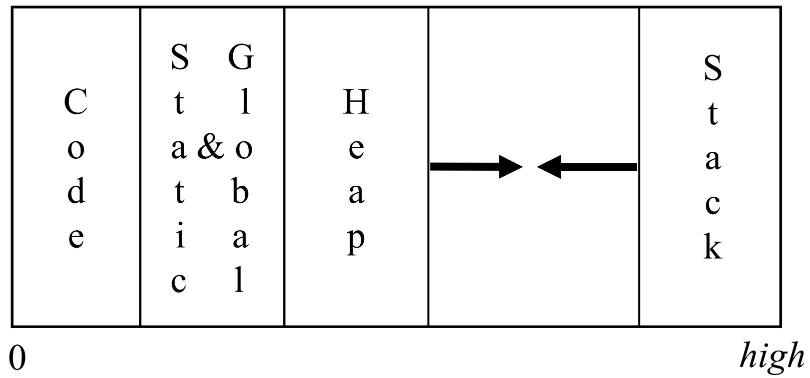
- Hard to extend

Static is easy

6.7 Managing Memory

Memory Layout

Placing run time data structures



Alignment & padding

- Both <u>languages</u> & <u>machines</u> have **alignment restrictions**
- Place values with identical restrictions next to each other
- Assign offsets from most restrictive to least
- Insert padding to match restrictions

Memory Model on Code Shape

Memory-to-memory model

- Compiler works within constraints of register set
- Each variable has a location in memory
- Register allocation becomes an optimization

Register-to-register model

- Compiler works with an unlimited set of virtual registers
- Only allocate memory for spills, parameters, & ambiguous values
  - Complex data structures (arrays) will be assigned to memory
- Register allocation is needed for correctness

Heap management

Allocate( ) & Free( )

- Implementing these requires attention to detail
- Watch allocation & free cost, as well as fragmentation
  - Many classic algorithms :
    first fit, first fit with rover, best fit

Implicit deallocation

- Humans are bad at writing calls to free()
- Major source of run-time problems
- Solution is to **automate deallocation**
- Reference counting + automatic free()
- Mark-sweep style garbage collection