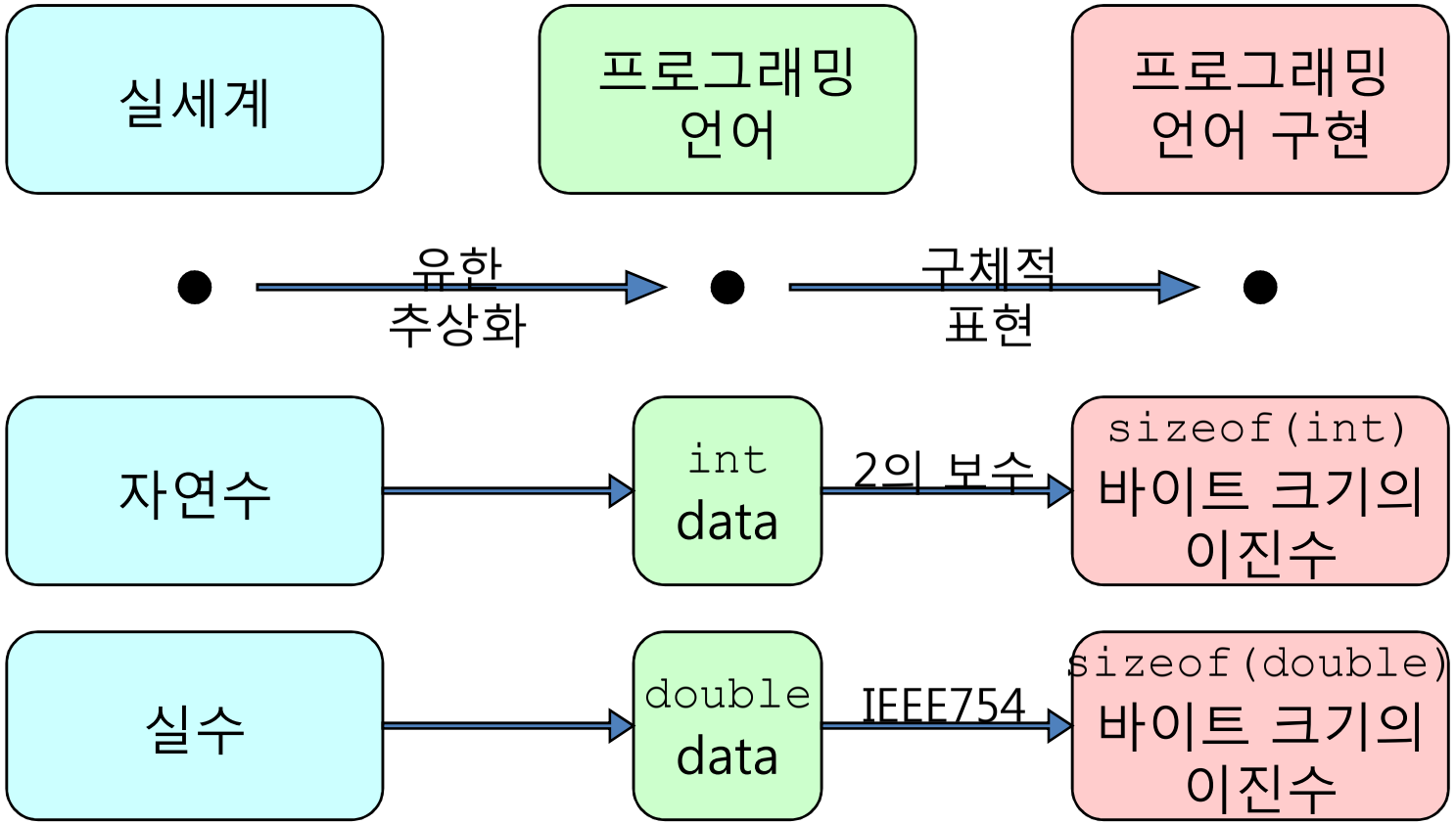
6 장
데이터 타입

데이터 타입과 타입 정보
타입 구성자
타입 동등성, 타입 검사
타입 변환
다형성

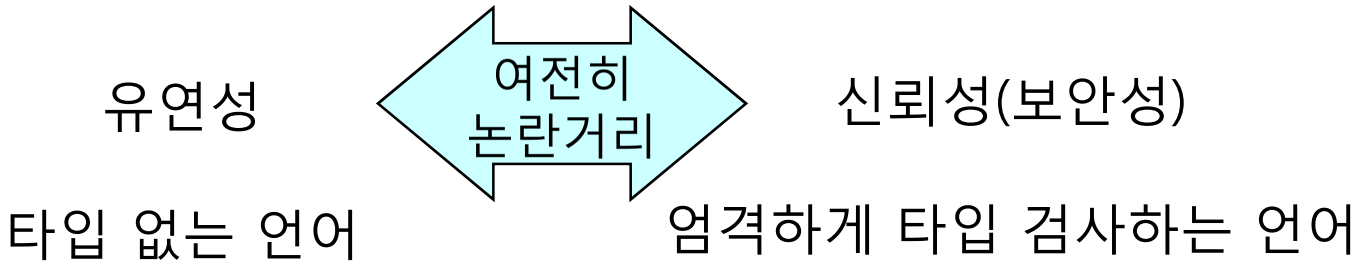
소개

- 타입은 추상화를 지원함



명시적인 타입 결정

- 프로그래밍 언어가 정적인 타입 검사를 지원해야 할지는 여전히 논란거리임



정적 타입 검사를 하는 이유

1. 수행 효율: 실행 시간에 타입 검사할 필요 없음
2. 번역 효율: 분리 컴파일이 가능함
3. 작성 용이성(코딩 효율): 컴파일 시간에 타입 에러 판별
4. 보안성과 신뢰성: 수행 오류를 줄임
5. 판독성: 문서화에 좋음
6. 모호성 제거: 다중적재된 이름 문제 해결
7. 설계 도구: 정적인 타입 검사
8. 인터페이스 일관성과 정확성: 추상 데이터 타입이 지원됨

데이터 타입의 정의

- 정의 1
  - 값들의 집합

  예) Java의 데이터 타입 **`int`** 는 다음 집합을 뜻함

  $V_{\texttt{int}} = \{ x \mid x \in \mathbf{Z}, -2{,}147{,}483{,}648 \le x \le 2{,}147{,}483{,}647 \}$
- 정의 2
  - 값들의 집합과 그에 상응하는 연산자들의 집합

  예) Java의 데이터 타입 **`int`** 는 다음 집합을 뜻함

  $V_{\texttt{int}} = \{ x \mid x \in \mathbf{Z}, -2{,}147{,}483{,}648 \le x \le 2{,}147{,}483{,}647 \}$

  그리고 다음의 연산자들의 집합을 뜻함

  $F_{\texttt{int}} = \{ +, -, *, \ldots \}$
- 연산자는 적절한 "피연산자"에 적용되어야 함

타입 검사와 타입 추론

- 타입 검사
  - 연산자(또는 서브프로그램)과 피연산자(또는 매개변수) 간의 타입 일관성을 결정하는 과정
- 타입 추론
  - 표현식에 타입을 부착하는 과정
- 타입 검사와 타입 추론은 서로 의존 관계임

타입 구성

- 타입 구성자
  - 기반 타입들은 매개 변수 없는 타입 구성자들임
  - 파생 타입들은 매개변수들을 가지는 타입 구성자들

- 타입 선언
  - 타입에 이름을 붙임

C의 예)

```
typedef int int10[10];
```

타입 시스템

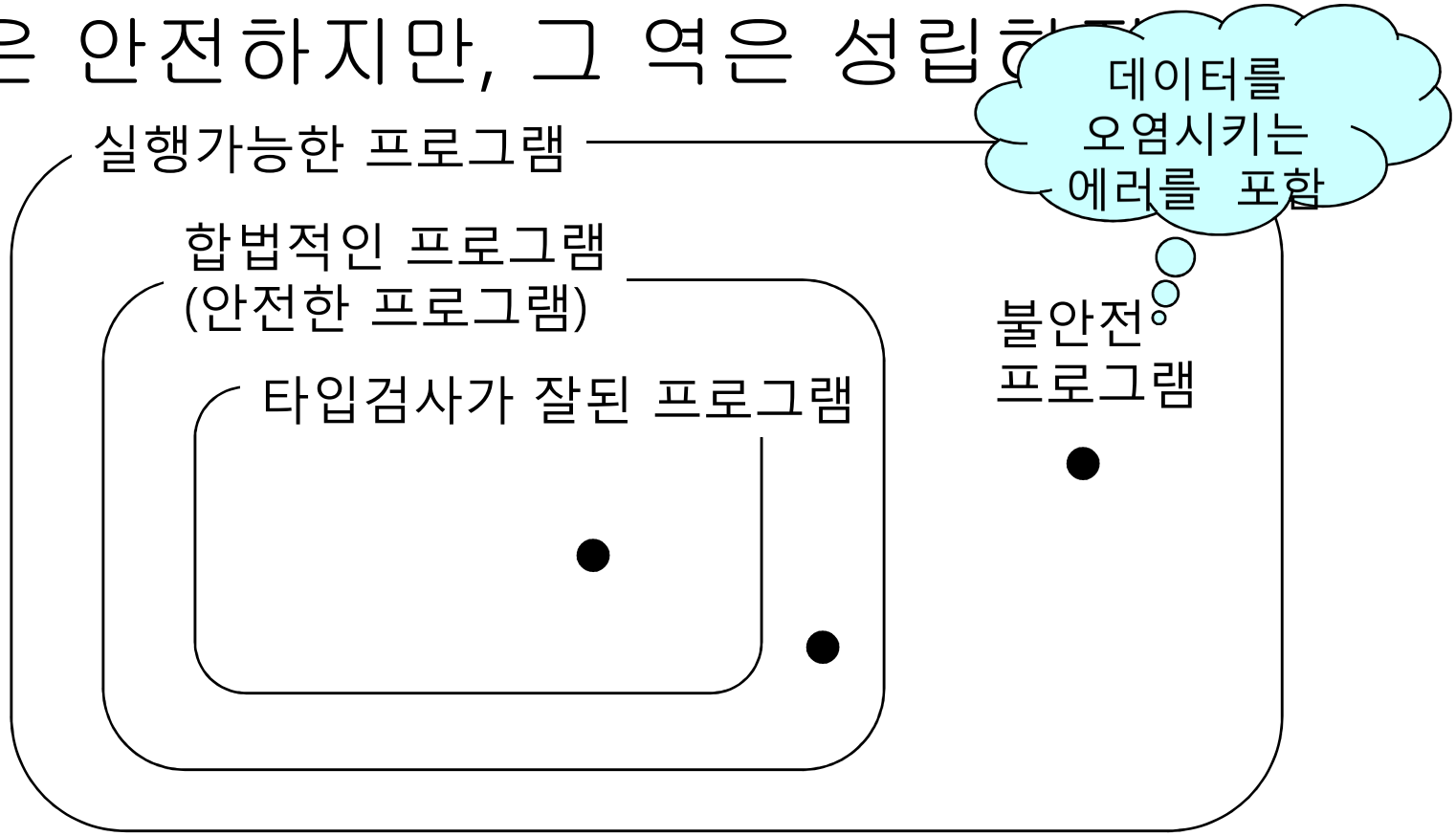
- 타입 동등성 알고리즘
  - 두 개의 타입을 비교하여 서로 같은지를 결정함
- 타입 시스템
  - 타입 구성 방법
  - 타입 동등성 알고리즘
  - 타입 추론과 타입 검사 규칙

언어의 타입 검사

- 엄격 결정 언어
  - 번역 시간에 모든 타입 에러를 찾아냄 (예: Ada)
- 해이 타입 결정 언어
  - 번역 시간에 타입 에러들을 찾아내지만, 몇몇 빠진 구석을 허용함 (예: C, C++)
- 타입 결여 언어
  - 정적인 타입 시스템이 없는 언어

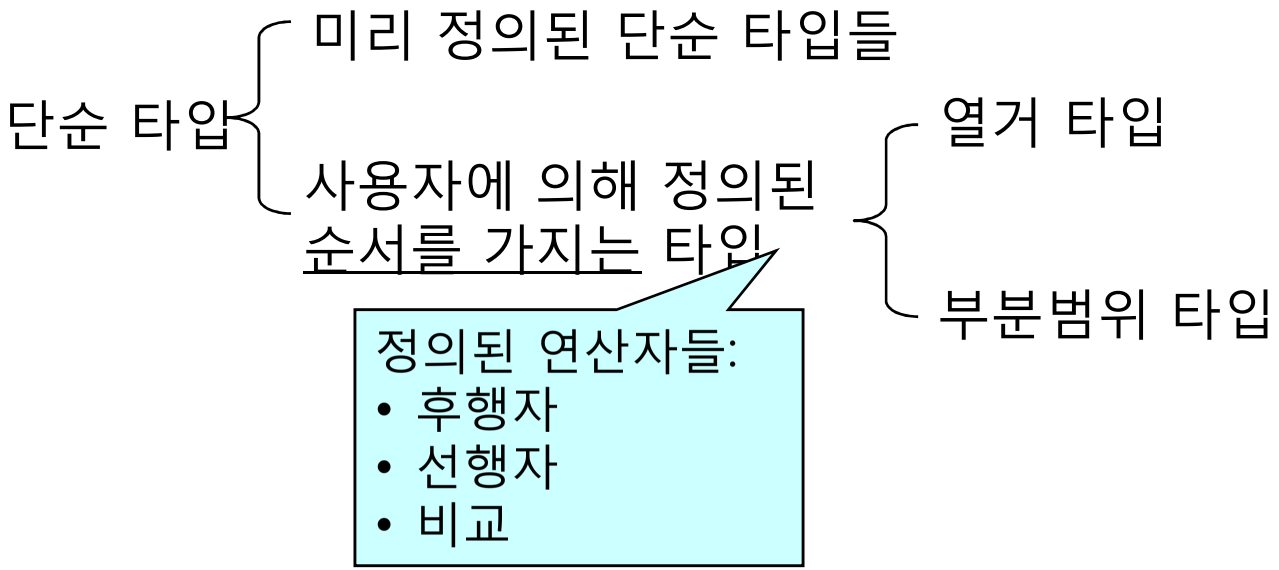
안전한 프로그램

- 타입 검사가 잘되어 있는 모든 프로그램은 안전하지만, 그 역은 성립하



단순 타입

- 단순 타입
  - 원자 타입
  - 다른 타입 구조를 포함하고 있지 않은 타입
- 단순 타입 분류



순서 타입들의 예

- C 에서의 열거 타입

```
enum Color { Red, Green, Blue };
```

- Ada 에서의 열거 타입

```
type Color_Type is (Red, Green, Blue);
```

- ML 에서의 열거 타입

```
datatype Color_Type = Red | Green | Blue;
```

- Java 는 열거 타입이 없음
- Ada 에서의 부분 범위 타입

```
type Digit_Type is range 0..9;
```

단순 타입에 대한 몇가지 노트

- 비교 연산자를 가지는 타입들이라고 전부 순서 타입은 아님

  예) Ada 의 부동소수점 숫자들의 부분 범위

```
type Unit_Interval is range 0.0..1.0;
```

- 대부분의 단순 타입들은 하드웨어에 의해 직접적으로 구현됨 (하드웨어 효율성이 중요함)

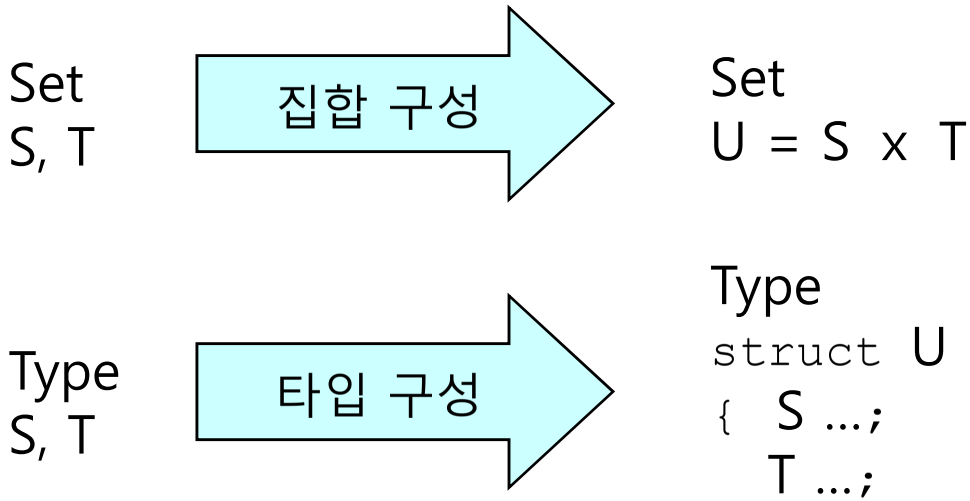
  예) Java의 정수 나누기는 소수점 부분을 절사함으로써 구현됨

```
(x / y) * y  + x % y ≡ x
```

음수일 수 있음

타입 구성자

- 타입들은 값들의 집합으로 간주할 수 있고, 타입 구성은 집합 구성으로 간주할 수 있음



데카르트 곱(Cartesian Product)

- 데카르트 곱은 쌍(pair)들의 집합으로

  $U \times V = \{ (u, v) \mid u \in U \text{ and } v \in V \}$

  p_1과 p_2의 투사함수(projection functions)들을 가짐 :

  $p_1 : U \times V \rightarrow U$

  $p_1 ((u, v)) = u$

  $p_2 : U \times V \rightarrow V$

  $p_2 ((u, v)) = v$

- 데카르트 곱들에 상응하는 타입 구성자들 : 레코드 (구조) 구성자들

예: C의 구조체 타입

```
struct IntReal     // IntReal = int x double
{
    int i;
    double r;
};
struct IntReal x = {1, 2.5};
                   // x = (1, 2.5) ∈ IntReal
x.i   // p1(x)
x.r   // p2(x)
```

요소 선택자 연산
(구조체 멤버 연산)

예: ML의 튜플 타입

```
(2, #"a", 3.14)
     (* 타입이 int * char * real 인 튜플 *)
#1
#2
#3
#3 (2, #"a", 3.14) = 3.14
```

요소 선택자

노트) 객체지향 언어에서는, 레코드 타입이 클래스로 확장 되었음

합집합

- 합집합의 종류
  - 비구별 합집합: 일반적인 집합 연산
  - 구별 합집합: 구별자 (태그) 가 추가됨
- 예: C 에서의 합집합
  - 비구별 합집합: 집합 그 자체
  - 구별 합집합: 구조체(struct) 안에 유니언(union)

```
union IntOrReal
{   int i;
    double r;
};
```

```
enum Disc {IsInt,IsReal};
struct IntOrReal
{   enum Disc which;
    union
    {   int i;
        double r;
    } val;
};
```

좀 더 안전한 합집합

- Ada 의 가변 레코드
  - 구별자(discriminant)와 값이 동시에 배정되어야 함

```
type Disc is (IsInt, IsReal);
type IntOrReal (which: Disc)
is
record
  case which is
    when IsInt =>  i:integer;
    when IsReal => r:float;
  end case;
end record;
...
x: IntOrReal := (IsReal,2.3);
```

- ML 의 데이터 타입
  - 열거자가 데이터 필드를 가질 수 있음
  - **case** 표현식을 이용한 패턴 매칭

```
datatype IntOrReal =
  IsInt of int |
  IsReal of real;
...
fun printIntOrReal x =
  case x of
     IsInt(i) => printInt i
   |
      IsReal(r) =>printReal
   r ;
```

합집합에 대한 노트

- 객체지향 언어에서는, 합집합은 상속으로 바뀜
- C++에서의 익명 합집합
  - C++ 에서는 태그 이름이 없는 익명 합집합을 지원함

```
enum Disc {IsInt,IsReal};
struct IntOrReal    // C++ only
{   enum Disc which;
    union
    {   int i;
        double r;
    };  // no member name here
};
```

부분집합

- 일반적으로 부분집합은 하위타입에 의해 지원됨
- Ada에서는, 하위타입과 부분범위 타입은 다름
  - 하위타입

```
subtype IntDigit_Type is integer range 0..9;
```

  - 새로운 타입 (이미 존재하는 타입의 부분범위)

```
type Digit_Type is range 0..9;
```

  - 가변 레코드 타입의 가변 부분을 고정시킴

```
subtype IRInt is IntOrReal(IsInt);
subtype IRReal is IntOrReal(IsReal);
```

- 객체지향 언어에서는, 상속은 서브타입 메커니즘임

배열들과 함수들

- 다음의 함수를 보자

  f: U → V

  만일 U 가 순서 타입이면, f 는 배열:

  U: 인덱스 타입

  V: 요소 타입

  f(i) ∈ V which i ∈ U
- 배열 크기 문제
  - 배열의 크기는 배열의 일부일 수도 있고 아닐 수도 있음

배열의 예: C/C++

- 배열의 크기는 배열의 일부가 아님
  - 배열 패러미터들을 보낼 때, 배열의 크기도 별도의 패러미터로 보냄

```
int array_max (int a[], int size);
```

  - 형식 인자 리스트에서 배열 표현은 포인터임
- C 와 C++ 배열 선언의 작은 차이:
  - C 에서 배열의 크기가 리터럴이나 상수식도 가능하며, C99와 그 이후 버전은 배열의 크기를 동적으로 설정할 수 있음
  - C++ 에서, 배열의 크기에서 배열의 크기가 리터럴이나 상수식도 가능

```
const int Size = 5;
int x[Size];       /* ok in C, ok in C++ */
int x[Size*Size]; /* ok in C, ok in C++ */
```

C99의 가변 길이 배열(VLA) - 중요

```
1. int main()
2. {
3.         int n = 5;
4.         int a[n];
5. }
```

- C++에선 템플릿 라이브러리가 강력하므로 필요없음

배열의 예: Java, Ada

- Java 에서는,
  - 배열의 크기는 배열의 일부
  - 배열이 동적으로 할당 가능함
- Ada 에서의 무제한 배열
  - 동적으로 크기가 정해지는 배열
  - 배열의 크기는 변수가 선언될 때 결정됨

```
// a part of Figure 6.3
...
int [] x = new int[u] ;
  // Dynamic array allocation
for (int i = 0; i < x.length;
   i++)
  x[i] = i;
...
```

```
type IntToInt is
  array (INTEGER range <>)
    of INTEGER;
...
declare
  x: IntToInt(1..size);
begin
  for i in x'range loop
    x(i) := i;
```

다차원 배열

- 다차원 배열 지원
  - 다차원 배열은 배열들의 배열로 시뮬레이션됨 (C, C++, Java)
  - 어떤 언어들은 별도의 다차원 배열을 지원함 (Ada, FORTRAN): Ada 에서는, `x(i,j)` ≠ `x(i)(j)`
- 다차원 배열의 메모리 할당
  - 행-우선 형태: 배열의 배열을 선언하는 일반적 형태
  - 열-우선 형태: FORTRAN

행-우선:

1	2	3
4	5	6

열-우선:

1	3	5
2	4	6

다차원 배열을 보내는 방법

- 차원 값을 보내는 방법
  - 배열의 크기가 배열의 일부라면, 문제 없음
  - 배열의 크기가 배열의 일부가 아니라면, 배열의 크기도 같이 보내져야 함
- 예) C/C++ 에서 2차원 배열을 보내야 할 때는 3가지 경우를 생각할 수 있음
  - 2차원의 크기를 다 미리 아는 경우
  - 1차원의 크기를 같이 보내야 하는 경우
  - 2차원의 크기를 모두 보내야 하는 경우

2차원의 크기를 미리 알고 있음 (중요)

```
#include <iostream>
using namespace std;

const int R = 3, C = 3;

int sum(int a[R][C])
{
    int sum = 0;
    for (int i = 0; i < R; ++i)
        for (int j = 0; j < C; ++j)
            sum += a[i][j];
    return sum;
}

main()
{
    int a[R][C] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    cout << "1+2+..+9 = " << sum(a) << endl;
}
```

1차원의 크기를 패스하는 경우 (중요)

```
#include <iostream>
using namespace std;

const int R = 3, C = 3;

int sum(int a[][C], int row)
{
    int sum = 0;
    for (int i = 0; i < row; ++i)
        for (int j = 0; j < C; ++j)
            sum += a[i][j];
    return sum;
}

main()
{
    int a[R][C] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    cout << "1+2+..+9 = " << sum(a, R) << endl;
}
```

각각의 차원들을 패스하는 경우 (중요)

```
#include <iostream>
using namespace std;

int sum(int a[], int row, int col)
{
    int sum = 0;
    for (int i = 0; i < row; ++i)
        for (int j = 0; j < col; ++j)
            sum += a[i*col+j];
    return sum;
}

const int R = 3, C = 3;

main()
{
    int a[R][C] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    cout << "1+2+..+9 = " << sum(&(a[0][0]), R, C) << endl;
}
```

2차원 배열에 대한 노트 (중요)

- 동일함과 차이점 (형식 인자 리스트에서)
  - 상수 N

    int a[N] ≡ int a[] ≡ int *a
  - 상수 M 과 N

    int a[M][N] ≡ int a[][N] ≡ int (*a)[N]
  - 주의할 것

    int a[][N] ≠ int **a
- C/C++ 에서의 선언
  - 선언이 사용 방법을 의미함

```
int *a[N];
int (*a)[N];
```

함수 타입

- 함수 타입은 포인터로 간주될 수 있음
- 언어의 예
  - C는 함수 타입 변수를 지원함: 역참조는 함수 타입 변수에 대해 암시적으로 지원됨 (중요)

```
typedef int (*IntFunction)(int);
int square(int x) { return x*x; }
IntFunction f = square;
int evaluate(IntFunction g, int value)
{ return g(value); }
```

  - Ada95 도 함수 타입 변수 지원함
  - Smalltalk 와 Java 같은 순수 객체지향 언어들은 함수 변수가 없음
    - 왜냐면 객체가 함수를 포함 하고 있으므로, 객체 변수로 충분함
  - C++ : 함수 포인터, 멤버함수 포인터, 함수 객체

포인터 (참조) 타입

- 포인터 (참조) 타입 구성자들
  - 이에 상응하는 집합 구성자들은 없음
  - 특정 타입을 가리키는 모든 주소들의 집합
  - 쓰레기 수집기가 사용되는 경우, 포인터는 암시적임 (Java, ML, Scheme 등등)
- 포인터에 대한 노트
  - 일부의 경우 (특히 C++), 참조와 포인터가 구별됨
    - 참조는 역참조만 가능한 상수 포인터

```
double r = 2.3, &s = r;
```

  - C 와 C++ 에서, 배열은 상수 포인터

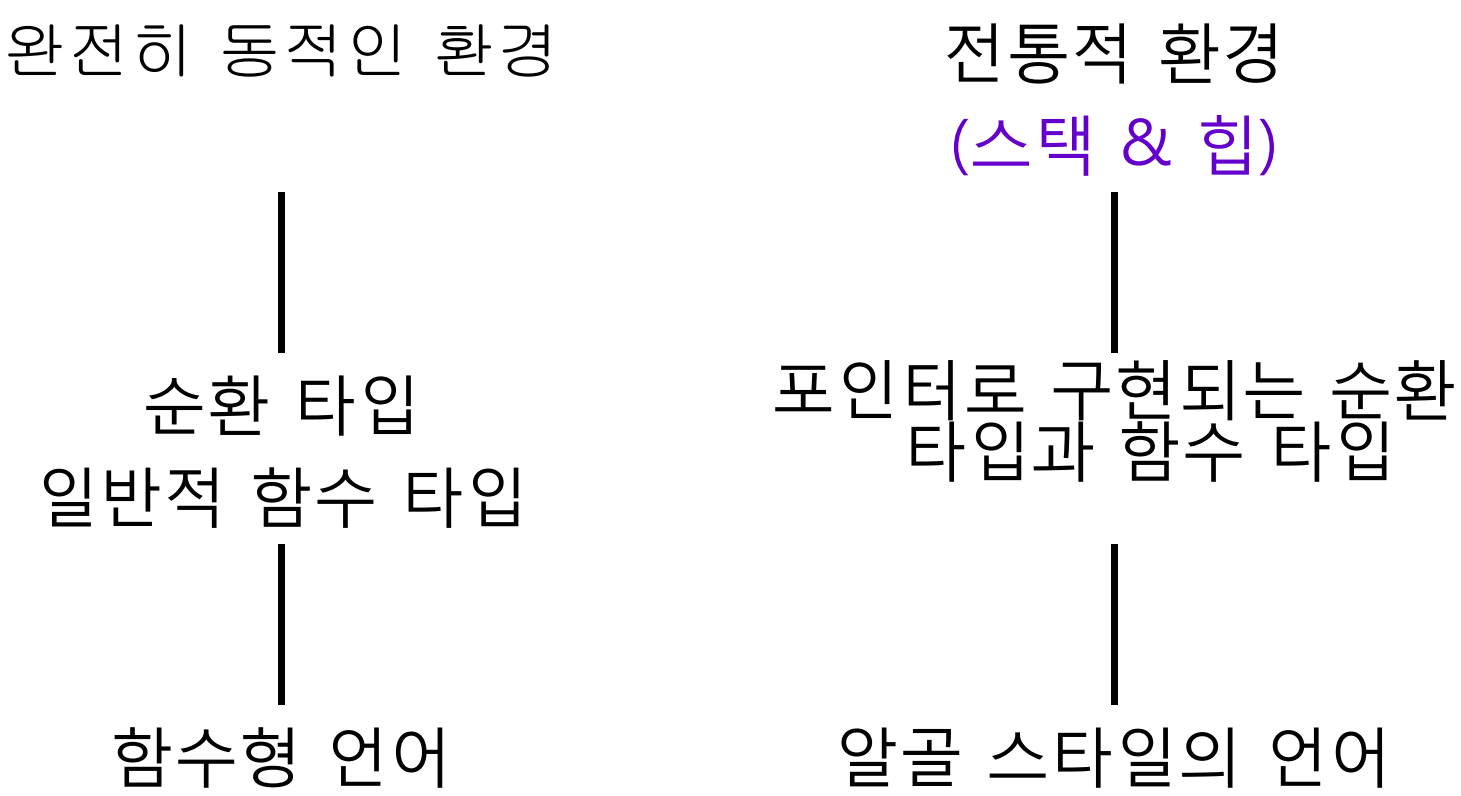
```
int a[] = {1,2,3,4,5};
a[2] ≡ *(a + 2) ≡ 2[a]   // Wow!
```

순환 타입 (Recursive Types)

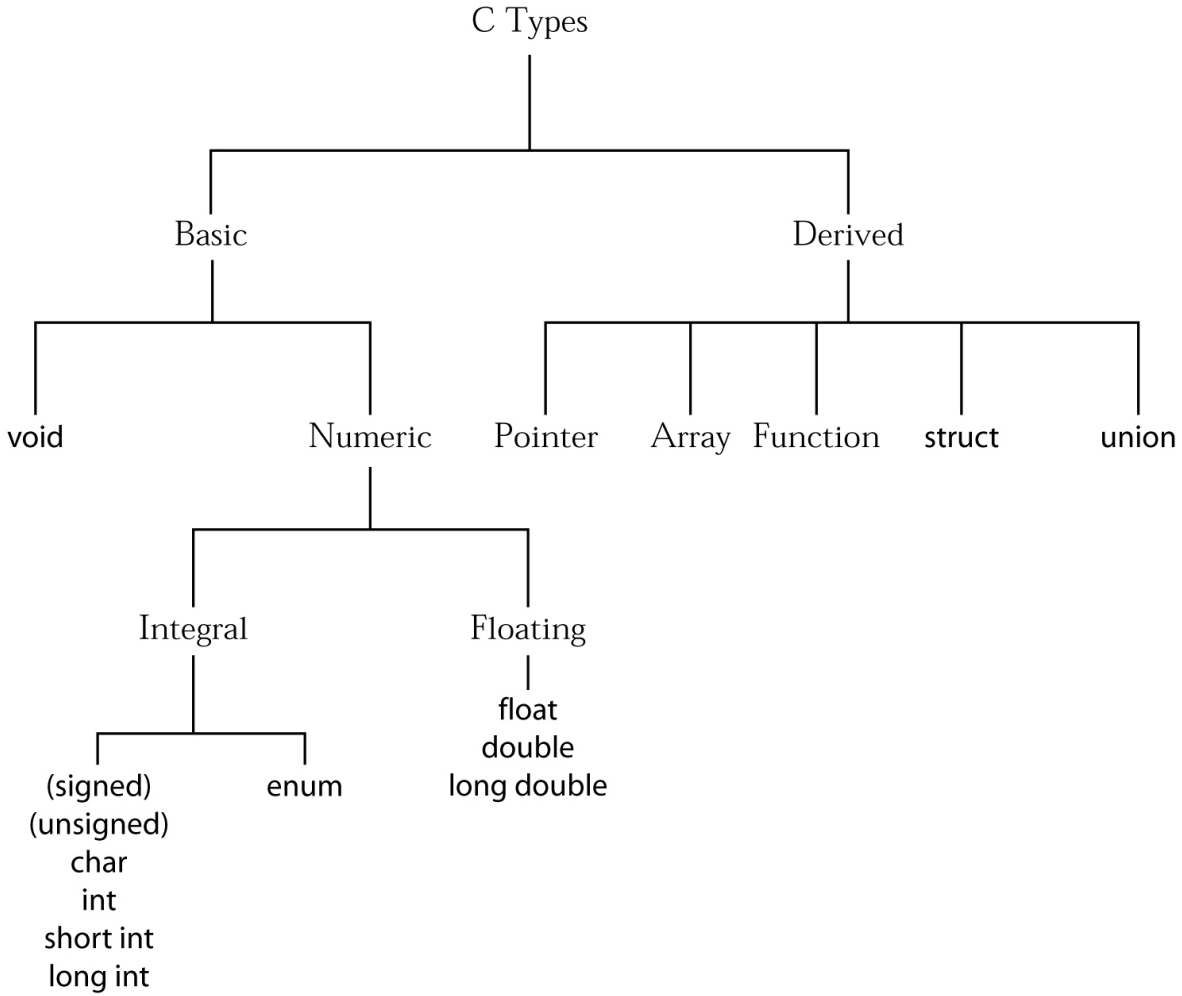
- 순환 타입
  - 선언에서 자신을 사용하는 타입
  - 순환 타입에서 포인터는 유용함
- 순환 타입의 크기 문제
  - 어떤 언어에서는 데이터 타입의 크기를 번역 시간에 다 알아야 함

```
struct CharList
{ char data;
  struct CharList next;
}; /* illegal! */

union CharList
{  enum { nothing } empty;
   struct
   { char data;
     union CharList next;
   } charListNode;
}; /* still, illegal! */

struct CharListNode
{ char data;
  struct CharListNode* next;
};
typedef   struct   CharListNode*
   CharList;
/* Now, legal */
```

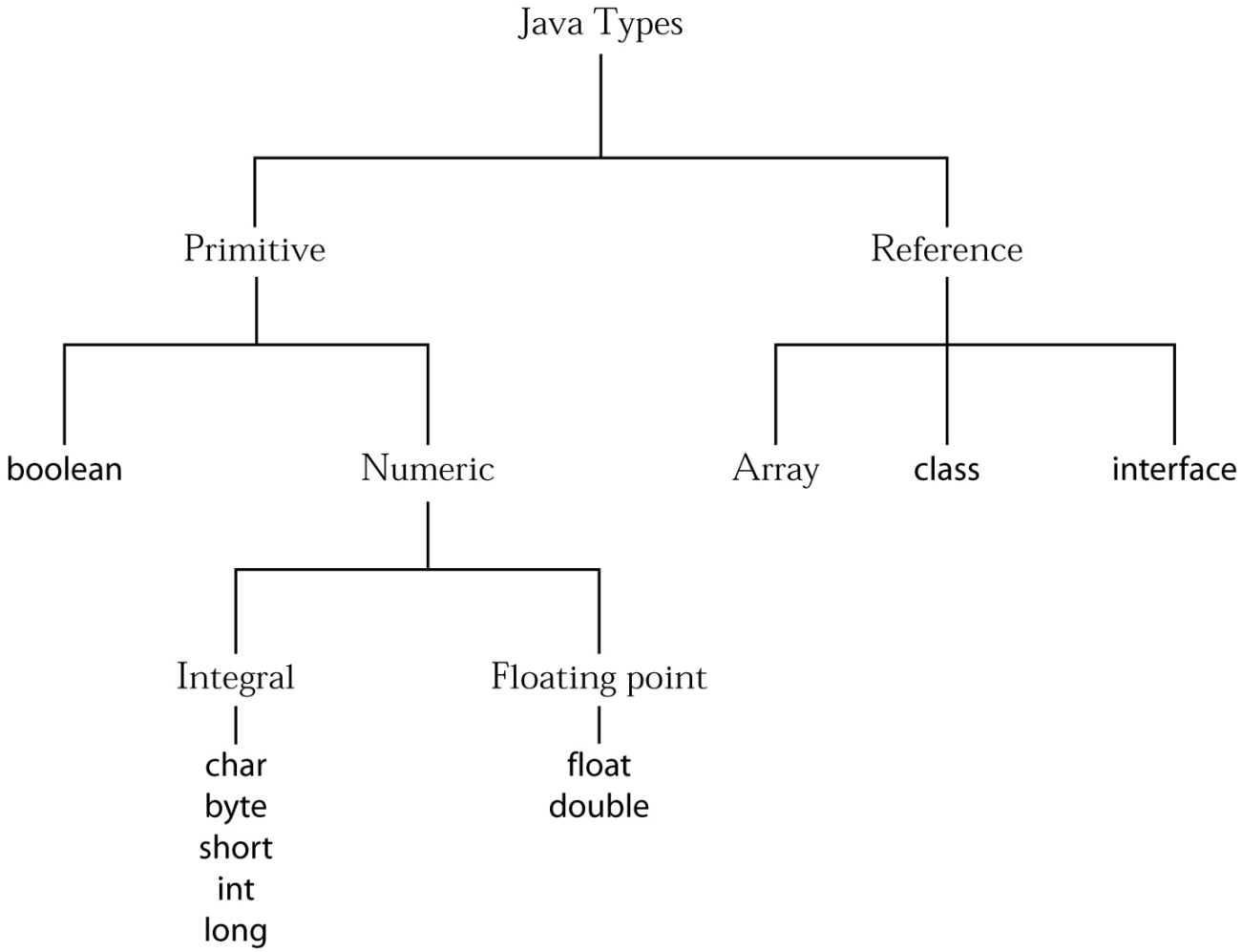
데이터 타입과 환경


완전히 동적인 환경
순환 타입
일반적 함수 타입
함수형 언어
전통적 환경
(스택 & 힙)
포인터로 구현되는 순환
타입과 함수 타입
알골 스타일의 언어

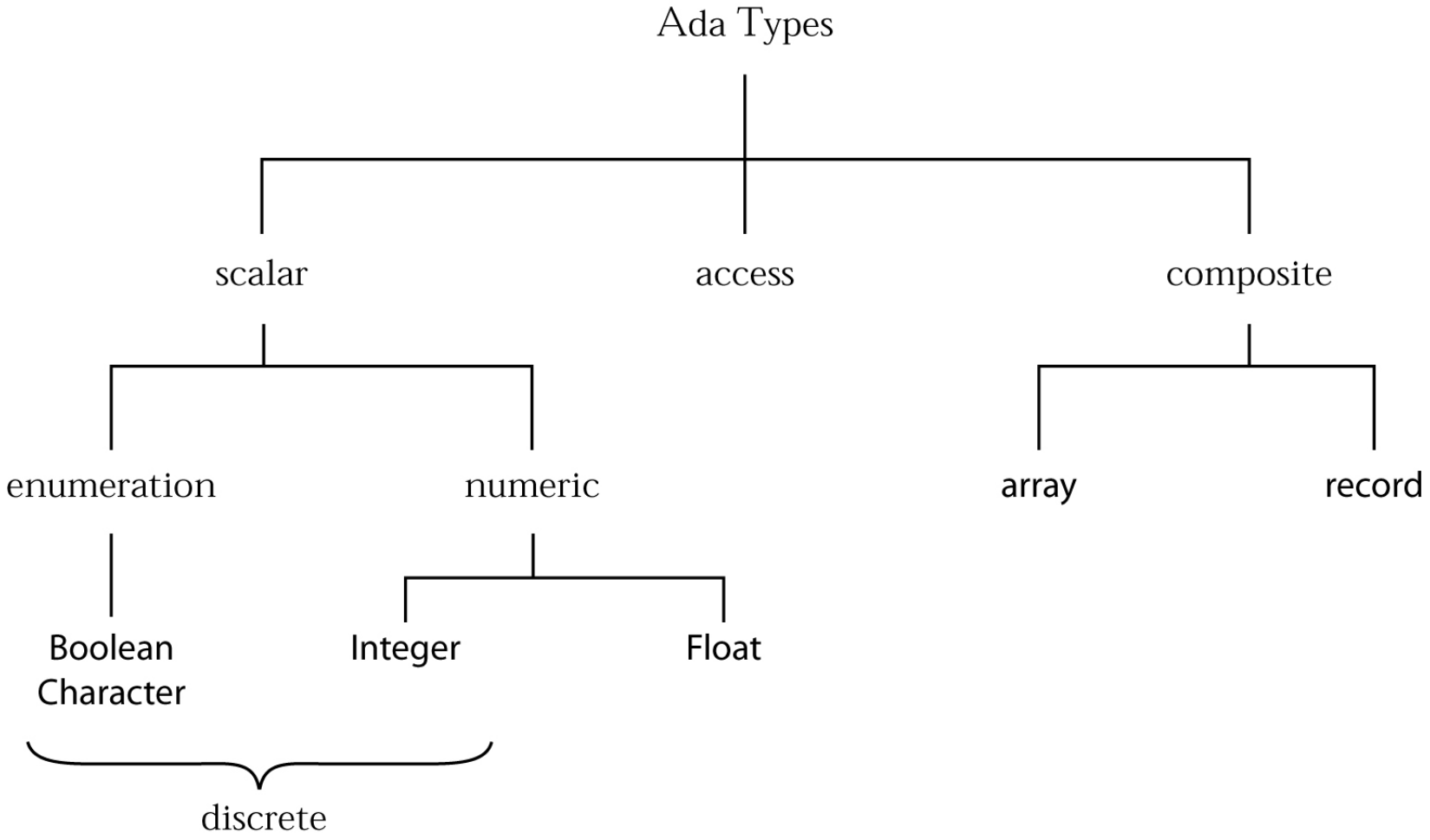
C 의 타입 구조


C Types
Basic
Derived
void
Numeric
Pointer
Array
Function
struct
union
Integral
Floating
(signed)
(unsigned)
char
int
short int
long int
enum
float
double
long double

Java의 타입 구조


Java Types
Primitive
Reference
boolean
Numeric
Array
class
interface
Integral
Floating point
char
byte
short
int
long
float
double

Ada의 단순화된 타입 구조



타입 동등성 (Type Equivalence)

- 구조적 동치 (Structural Equivalence)
  - 두 개의 타입은 같은 구조를 가지면 동일함
  - 같은 구조란, 같은 타입 생성자와 같은 요소 타입을 의미함
- 이름 동치 (Name Equivalence)
  - 두 개의 타입은 같은 이름을 가지면 같다고 봄
- 선언 동치 (Declaration Equivalence)
  - 같은 이름에서 파생된 타입은 같다고 봄

복잡하게 만드는 요소들

- 배열 크기
  - 배열 타입들의 크기는 배열 타입의 일부로 간주될 수 있음
  - 만일 그렇다면, 배열 패러미터의 타입 검사는 문제의 소지가 있음
- 레코드 타입의 필드 명
  - 다음의 두 구조체 **RecA** 와 **RecB** 는 구조적으로 같으나 (**char** X **int**), 필드 명 때문에 실제로는 다름

```
struct RecA
{   char x;
    int y;
};
```

```
struct RecB
{   char a;
    int b;
};
```

언어 예: Ada

- Ada
  - 완전히 이름 동치 방식임
    - 변수 a 와 b 는

      ```
      a, b: array (1..10) of integer;
      ```

      타입 동치가 아님, 그 이유는 위의 선언이 다음과 같이 해석되기 때문임

      ```
      a: array (1..10) of integer;
      b: array (1..10) of integer;
      ```
  - 새로운 타입과 서브타입은 서로 다름
    - 새로운 타입

      ```
      type Age is new integer;
      ```
    - 서브타입: 원래의 타입과 호환 가능

      ```
      subtype Age is integer;
      ```

언어 예: C

- 구조체와 유니언에 대해서는 이름 동치
- 다른 경우는, 구조적 동치
- '**typedef**' 은 새로운 타입을 맨드는 게 아니라, 별명을 만듬
- 배열 크기는 배열 타입의 일부가 아님

```
struct A = C
≠ struct B

P = R ≠ R

S = T

Age = int

F = G
```

```
struct A
{ char x;
  int y;


struct B
{ char x;
  int y;
};

typedef struct A C;
typedef C* P;
typedef struct B * Q;
typedef struct A * R;
typedef int S[10];
typedef int T[5];
typedef int Age;
typedef int (*F)(int);
typedef Age (*G)(Age);
```

언어 예: 다른 것들

- 파스칼 (Pascal)
  - 새로운 타입 생성자는 새로운 타입을 만듬
  - 존재하는 타입 이름을 위한 새로운 타입 이름은 동등함
- 자바 (Java)
  - '**`typedef`**' 구성물이 없음
  - 클래스와 인터페이스 선언은 새로운 타입 이름을 만듬
  - 상속 체계를 가지고 있음 (서브타입)
  - 배열에 대해 구조적인 동치를 제공함
- ML
  - '**`datatype`**' 은 새로운 타입을 만듬
  - '**`type`**' 구성물은 존재하는 타입에 대한 별명임

타입 검사

- 타입 검사의 종류
  - 동적 타입 검사
  - 정적 타입 검사
    - 약하게 타입을 검사하는 언어들
    - 엄격하게 타입을 검사하는 언어들: 모든 타입 에러는 실행 시간 이전에 다 잡힘

노트) 어떤 언어들은 동적인 또는 정적인 타입 검사가 사용될지 그렇지 않을지를 일부러 지정하지 않고 놔두기도 함

타입 검사의 예

- C/C++
  - 정적인 타입 검사
  - 엄격한 타입 검사는 아님: 타입 변환 가능
- Scheme
  - 동적인 타입 검사 그러나 엄격함
  - 미리 정의된 타입 테스트 함수들: **`number?`**, **`symbol?`**, …
- Ada
  - 엄격한 타입 검사
  - 배열의 한계 검사는 실행 시간에 함: 예외 발생

타입 검사와 타입 추론

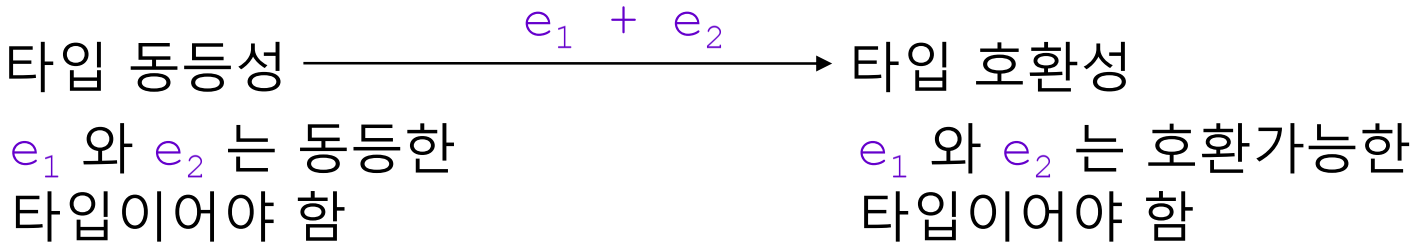
- 타입 추론
  - 주어진 표현식의 일부에서 그 표현식의 타입을 추론하는 과정
- 타입 검사와 타입 추론
  - 타입 검사 규칙과 타입 추론 규칙은 서로 얽혀있음

    $e_1 + e_2$

    전체 표현식의 타입 에러를 검사하기 위해서는, 그 표현식의 일부에서 타입을 추론해야 함
  - 타입 검사 규칙은 타입 동등성 알고리즘과 긴밀히 연결된 타입 추론 규칙인 셈임
  - 명시적인 선언은 타입 검사에 도움이 되지만, 프로그래머에게 의무적인 건 아님

타입 호환성 (Type Compatibility)

- 타입의 옳고 그름에 대한 기준을 완화함



  배정 호환성 (assignment compatibility)을 위해서는, 정보 보존이 중요한 요구조건임

- 언어의 예
  - Ada: 부분범위 타입은 원래의 기본 타입과 호환가능함
  - C, Java: 수치적인 타입들은 호환가능함

암시적인 타입들

- 암시적 타입들
  - 객체들의 타입이 명시적으로 선언되지는 않으나, 암시적으로 추론되는 경우
- 예제들
  - C: 변수와 함수들의 타입이 암시적으로 **int** (K&R 버전의 경우)

```
x;  /* 암시적으로 int */
f(int x) { return x + 1; } /* int 반환 */
```

  - Pascal: 상수는 그 리터럴 값에서 암시적으로 타입 지정됨

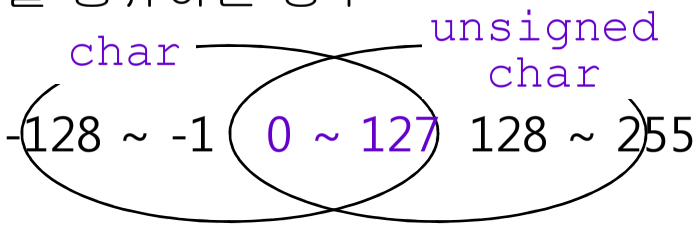
```
const PI = 3.14159;  (* 암시적으로 real *)
```

- 노트
  - 대부분의 언어에서 리터럴은 암시적으로 타입 지정됨
  - 만일 부분범위 타입이 허용된다면, 리터럴은 여러 타입들을 가질 수 있음 : **0** 은 **integer** 이면서 만일 다음과 같은 선언이 있다면 **Digit_Type** 일 수도 있음

```
type Digit_Type is range 0..9;
```

겹치는 타입(Overlapping Types)

- 값들은 겹치는 타입을 통해 여러 개의 타입으로 지정될 수 있음 – 두 개의 타입들이 같은 값들을 공유하는 경우
  - C 언어,



  - 어떤 언어에서는 실행시간 타입 검사가 적용됨
    - Ada 에서는, **CONSTRAINT_ERROR** 가 발생할 수 있음.

```
subtype SubDigit is integer range 0..9;
subtyep NegDigit is integer range -9..-1;
x: SubDigit;
y: NegDigit;
...
x := y
```

  - 겹치는 타입을 가지는 값들은 코드를 단순화하는 데 도움이 됨 – C 언어에서, 0은 정수거나 포인터 타입일 수 있음

공유되는 연산자(Shared Operators)

- 공유되는 연산자란 다중적재되는 연산자를 의미함
- 공유되는 연산자를 구별하려면?
  - 인자(Argument)의 타입
    - Java에서 만일 피연산자의 크기가 정수보다 작으면 정수 연산자가 선택됨 (C/C++도 마찬가지)

```
short x = 2;
x = x + x;    // 에러?! → 에러임
x = (short) x + x;    // 맞음
```

  - 반환 값의 타입 (C++나 Java에선 안됨)
    - Ada 의 예

```
function "+"(x,y: Digit_Type) return Digit_Type is ...
a: Digit_Type := 5+7;
```

C/C++ 두 가지 예제 #1

```
#include <iostream>
using namespace std;
int main()
{
   unsigned int x = 10;
   if (x < -1) cout << "작다₩n";
   else cout << "크다₩n";
   return 0;
}
```

C/C++ 두 가지 예제 #2

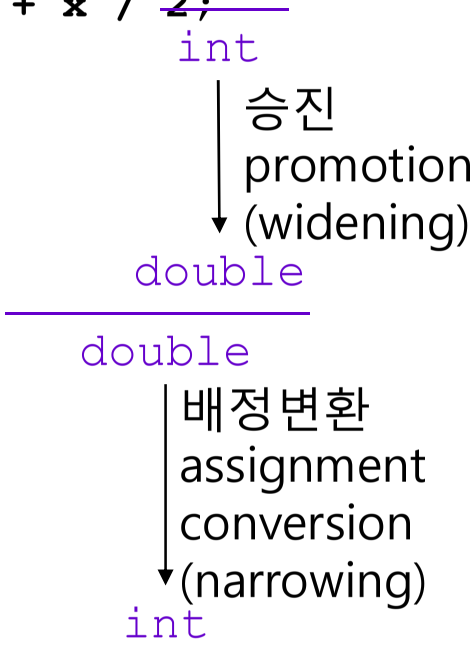
```
#include <iostream>
using namespace std;
int main()
{
    char x = 'A';
    cout << x << endl;
    cout << x+1 << endl;
    cout << x++ << endl;
    return 0;
}
```

타입 변환 (Type Conversion)

- 타입 변환의 분류
  - 표현법에 따라
    - 암시적 변환 (강압; coercion)
    - 명시적 변환 (캐스팅)
  - 타입의 크기에 따라
    - 확장 변환
    - 축소 변환 – 데이터의 손실이 생길 수 있음

- C 의 예

```
int x = 3;
x = 2.3 + x / 2;
```



타입 변환의 노트

- 명시적 변환의 장점
  - 타입 변환이 정확히 문서화됨
  - 번역기가 다중적재 해결을 더 쉽게 함

```
double max(int, double); // max #1
double max(double, int); // max #2
...
max(2, 3);           // calls what?
max(2, (double)3); // calls max #1
```

- 구조 캐스팅
  - 구조 타입의 크기가 동일해야 함
  - 번역기는 단순히 메모리를 재해석할 뿐

포괄적 (Generic) 포인터 (임의의 포인터)

- 포괄적 포인터는 아무 거나 가리킬 수 있음
  - 익명 포인터 (**`void*`** in C)

  - C에서는 양쪽 변환이 암시적으로 가능함
  - C++에선 변환 1은 암시적일 수 있으나, 변환 2는 명시적이어야 함

변환을 위한 라이브러리 함수

- Ada 의 경우
  - 문자 타입의 어트리뷰트 함수

```
character'pos('a')    -- returns 97
character'val(97)     -- returns 'a'
```

- Java 의 경우
  - Integer 클래스의 변환 함수들

```
String s = Integer.toString(12345);
int i = Integer.parseInt("54321");
```

엄격한 타입 검사의 틈새

- 엄격한 타입 검사에서 비구별 집합 구현은 틈새가 있음
- 예) C++에서 내부적으로 부울값 **true** 에 대해 어떤 정수값을 사용하는지 검사하는 경우

```
union
{   char c;
    bool b;
} x;
x.b = true;
cout << static_cast<int>(x.c) << endl;
```

다형적인 타입 검사

- 기존의 타입 검사

  선언들 ⟶ 타입 정보를 모음

  이름-타입

- Hindley-Milner 타입 검사

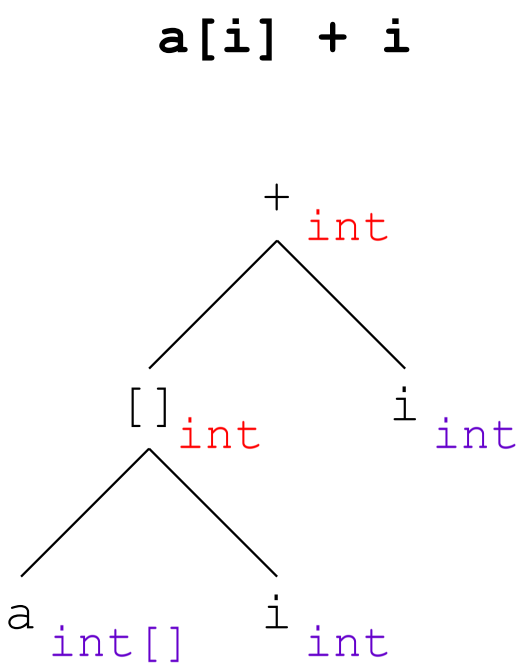
  이름의 사용 ⟶ 타입 정보 추론

  이름-가장 일반적인 타입

  (주된 타입)

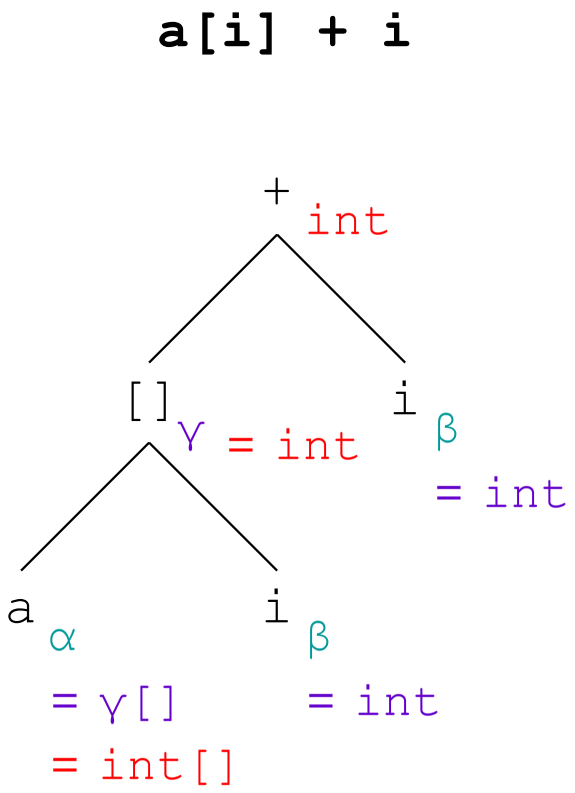
기존의 타입 검사기

1. 구문 트리(Syntax Tree)
2. 노드를 타입값으로 지정
3. 타입 검사 (내부적인 변환은 허용)



Hindley-Milner 타입 추론

1. 구문 트리(Syntax Tree)
2. 타입 변수 배정
3. 타입 제한사항 수집
4. 가장 일반적인 타입 판별



타입 추론에 대한 노트

- Hindley-Milner 타입 추론의 경우
  - 모든 타입 변수들은 같은 타입 값으로 구체화 되어야 한다 (구체화)
  - 두 개의 타입 변수들은 이름을 사용하는 용례에 따라 단일화되어야 한다 (단일화)

단일화 (Unification)

- 단일화 (Unification)
  - 두 개의 타입이 단일화될 수 있는지를 검사
- 단순한 단일화 알고리즘이 포함하는 세 가지 경우
  - 타입 변수는 어떤 것이든 임의의 타입 표현과 단일화
  - 임의의 두 개의 타입 상수(int 나 char)는 서로 동일해야만 단일화됨
  - 임의의 두 개의 타입 구성 (배열이나 구조체같은 타입 구성자)은 서로 동일한 타입 구성자가 적용된 것이고 그 구성요소 타입들 전부가 단일화될 때만 단일화

U: type expression × type expression → boolean

$U(v, e) = \text{true}$

$$U(c_1, c_2) = \begin{cases} \text{true} & \text{if } c_1 = c_2 \\ \text{false} & \text{otherwise} \end{cases}$$

$$U(c_1(e_{11},\ldots,e_{1n}), c_2(e_{21},\ldots,e_{2m})) = \begin{cases} U(e_{11},e_{21}) \text{ and } \ldots \text{ and } U(e_{1n},e_{2n}) & \text{if } c_1 = c_2 \text{ and } n = m \\ \text{false} & \text{otherwise} \end{cases}$$

다형적 타입 검사

- Hindley-Milner 타입 시스템에서는 다형적 타입 검사를 구현했음
- 다형성: 하나의 이름이 동시에 하나 이상의 타입을 가지는 것 (중요!)
  - ad hoc polymorphism (overloading; 다중 적재)
  - parametric polymorphism (Hindley-Milner; 템플릿)
    - implicit parametric polymorphism (암시적 매개변수 다형성)
    - explicit parametric polymorphism (명시적 매개변수 다형성)
  - pure polymorphism (subtype polymorphism; 상속 및 가상함수)

다형성의 장점

- 단형적인 함수 (in C)

```
int max(int x, int y)
{ return x > y ? x : y; }
```

  - 문제 1: `max` 는 `int` 타입에만 사용 가능.
  - 문제 2: `int` 타입이 `>` 연산자를 가지고 있어야 함

```
int max(int x, int y, int(*gt)(int,int))
{ return gt(x,y) ? x : y; }
```

  - 위와 같이 해도 문제 1 은 해결되지 않음.

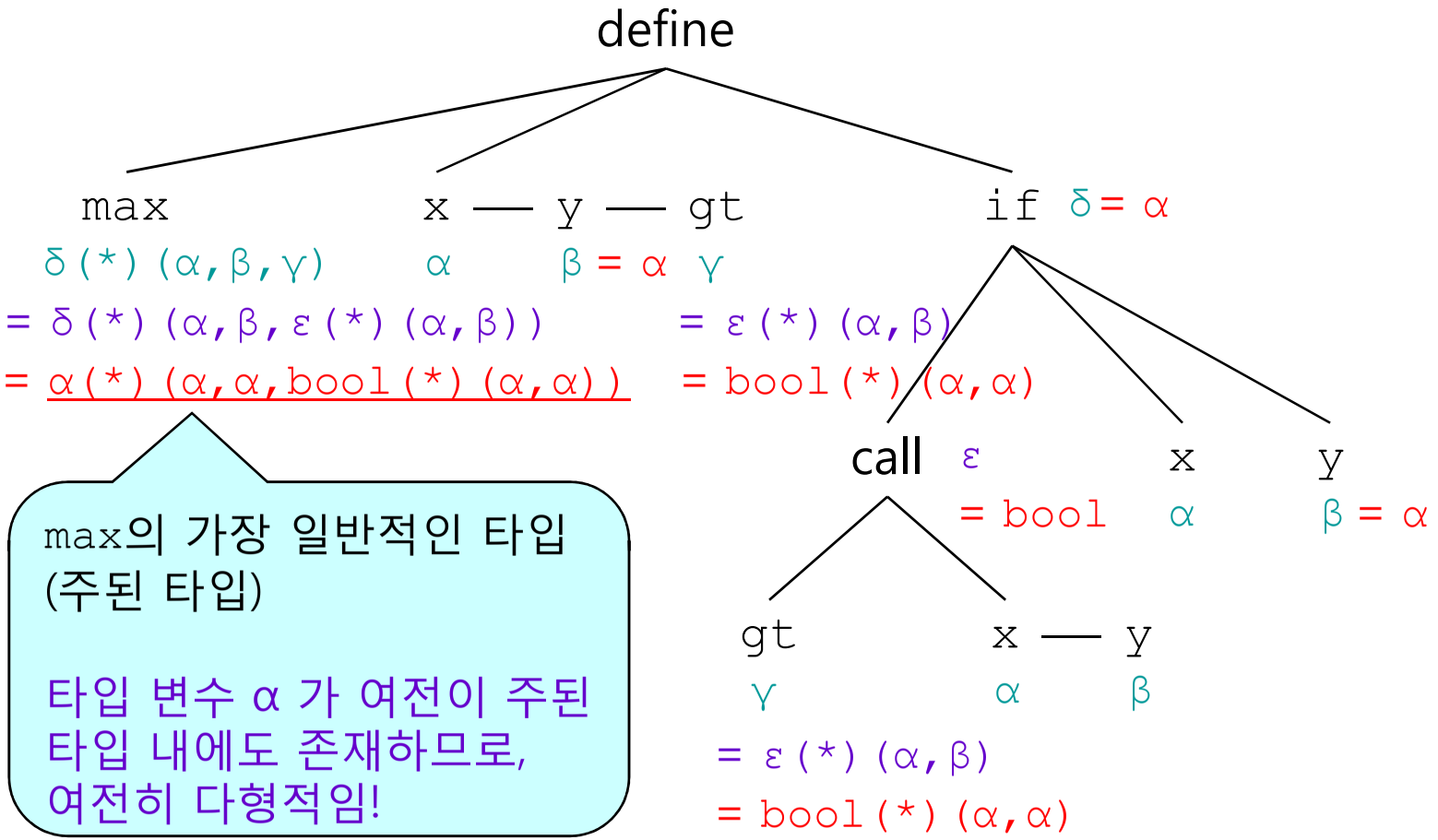
- 다형성을 가진 함수 (ML 의 예)

```
fun max(x,y,gt) = if gt(x,y) then x else y;
```

  - `x`, `y` 의 타입과 `gt` 는 지정되지 않았음에 주목

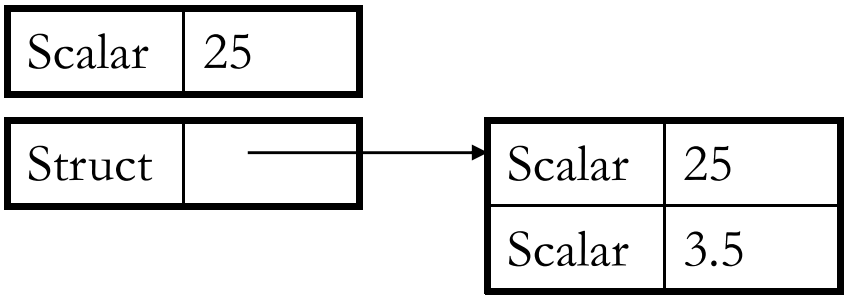
더 복잡한 예

```
fun max(x,y,gt) = if gt(x,y) then x else y;
```



다형적 타입을 구현하는 방법

- 확장 (Expansion)
  - 서로 다른 타입에 대해 코드의 새로운 복사본을 생성
  - 코드 크기가 커지는 문제
- 박싱(Boxing; 책에선 '상자'로 번역; 중요!) 및 태깅 (Tagging)
  - 모든 데이터 종류마다 태그를 붙임 (즉, 스칼라 데이터인지 아니면 구조적 데이터인지...)
  - 태깅과 역참조의 오버헤드가 있음



다형적 데이터 타입은 어떤가?

- 다형적 데이터 타입을 구성하기 위해서는, 타입 변수가 있어야 함 → 명시적 다형성!
- C에서 다음처럼 가정하면

```
typedef struct StackNode
{   ?? data;
    struct StackNode *next;
} *Stack;
```

node 데이터의 타입이 선언되야 함
타입를 지정하는 변수 메커니즘이 있어야 함

- In ML,

```
datatype 'a Stack = EmptyStack
                  | Stack of 'a * ('a Stack)
val x = Stack(3,EmptyStack);
```

타입 구성자

타입은 순환적으로 정의가 가능함

데이터 구성자

C++ 의 명시적 다형성 (중요!)

- 템플릿 구조

```
template <typename T>
struct StackNode
{ T data;
  StackNode<T> * next;
};
template <typename T>
struct Stack
{ StackNode<T> *
  theStack;
};
```

- 템플릿 함수

```
template <typename T>
T top (Stack<T> s)
{ return
     s.theStack->data;
}
```

- 템플릿 사용

```
Stack<int> s;
s.theStack = new
  StackNode<int>;
s.theStack->data = 3;
s.theStack->next = 0;
int t = top(s);
  //  t is now 3
```

템플릿에 대한 노트 (중요!)

- 템플릿 함수를 사용할 때, 타입 패러미터는 기술할 필요 없음
- 템플릿 구조의 변수를 사용할 때에는, 타입 패러미터를 기술해야 함

템플릿 함수 **max**

- ML 에서는 다음과 같은 다형성 함수가 있었음

```
fun max(x,y,gt) = if gt(x,y) then x else y;
```

- 이런 함수는 C++에서 다음과 같이 정의됨

```
template <typename T>
T max (T x, T y, bool (*gt)(T,T))
{ return gt(x,y) ? x : y ;
}
bool gti (int x, int y)
{ return x > y; }
int larger = max(2,3,gti);  // larger is now 3
```

다른 템플릿 함수 **max**

- 다음의 **max** 는 타입 **T** 가 **>** 연산자가 있다고 가정함

```
template <typename T>
T max (T x, T y)
{ return x > y ? x : y ;
}
int    larger_i = max(2,3);       // OK
double larger_d = max(3.1,2.9);   // OK
Stack<int> s, t, u;
u = max(s,t);
  // 에러! Stack 타입에 대해서 > 가 정의되어 있지 않음
```

- 이러한 형태의 다형성을 암시적 (또는 묵시적) 제약 매개변수 다형성이라 함

Ada 의 제니릭 패키지 (포괄적 패키지)

- 패키지 스펙

```
generic
  type T is private;
package Stacks is
  type StackNode;
  type Stack is access
   StackNode;
  type StackNode is
  record
    data: T;
    next: Stack;
  end record;
  function top( s: Stack )
   return T ;
end Stacks;
```

- 패키지 바디

```
package body Stacks is
  function top( s: Stack )
   return T is
  begin
    return s.data;
  end top;
end Stacks;
```

- 패키지 사용

```
package IntStacks
  is new Stacks(integer);
use IntStacks;
s: Stack := new StackNode;
t: integer;
...
s.data := 3;
s.next := null;
t := top(s); -- t is now 3
```

Ada의 제네릭 함수 (포괄적 함수)

- 함수 스펙

```
generic
  type T is private;
  with function gt(x,y:T)
   return boolean;
function max (x,y:T) return T;
```

- 함수 몸체

```
function max (x,y:T) return T
   is
begin
   if gt(x,y) then return x;
   else return y;
   end if;
end max;
```

- 함수 사용

```
with max; -- import
...
function maxint is new
   max(integer,">");
i: integer;
...
i := maxint(1,2);
```

명시적 제약
매개변수
다형성