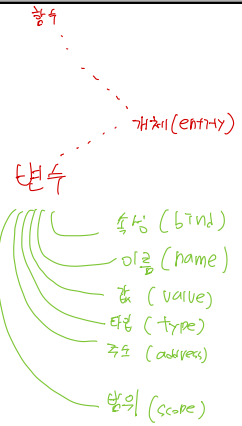
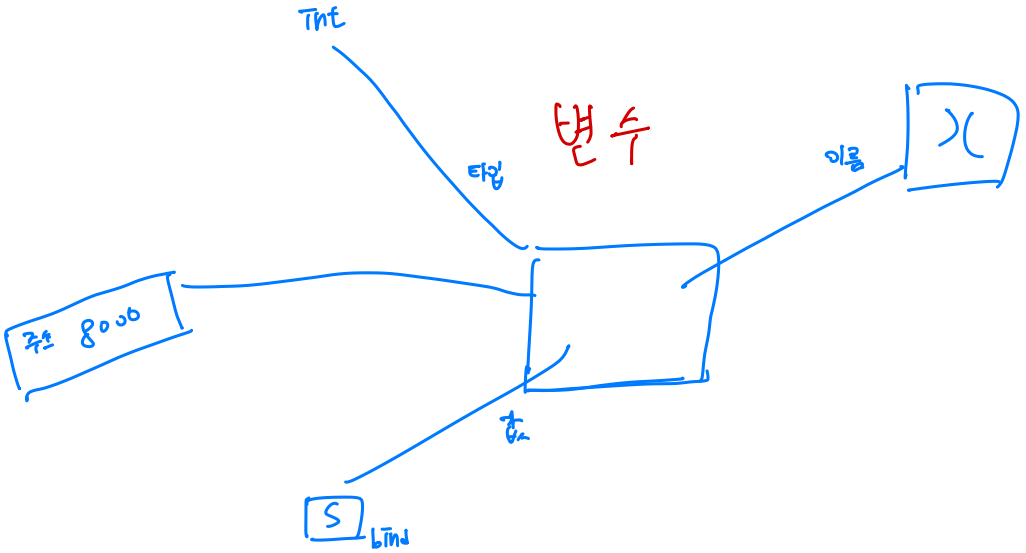
시험 ✱ 변수의 6가지 속성을 적으시오 ✱



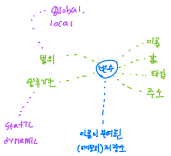
Variable, binding, scope

```
int x = 1;
int main() {
    int x;
    x = 5;
}
```



목차

- Variable
- Binding
  - Type binding
  - Storage binding
- Scope
  - Static scoping
  - Dynamic scoping
- Block



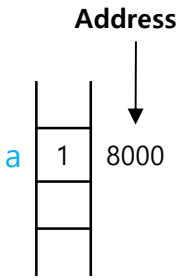
Variable

Variable(변수)

- 값(**value**)을 저장하기 위한, 이름(**name**)이 부여된, 저장 공간으로, 특정 메모리 주소(**address**)와 연결되며 타입(**type**)이 부여되고, 추가적으로 **scope**(변수가 사용될 수 있는 코드 범위)과 **lifetime**(변수가 저장공간에 연결(**binding**)된 동안의 시간)이라는 속성을 갖는다
- 변수의 타입으로부터, 컴파일러는 변수에 저장되는 값의 표현 방식, 가능한 값의 범위, 허용 가능 연산을 결정 → 예) Java에서 int score;로부터 score에 저장 가능한 값의 범위는 -2147483648 ~ 2147483647이며 저장되는 값은 2의 보수로 표현되며, score에 저장된 값에 적용 가능한 연산은 덧셈, 뺄셈 등임을 알 수 있음
- Scope은 static scoping 혹은 dynamic scoping으로 결정 가능하며 대부분 PL은 static scoping 사용
- 변수의 type은 컴파일 시점에 결정되는 static typing(예: C, Java)과 실행 시점에 결정되는 dynamic typing(예: Python, Javascript)이 있음
- 변수는 그 lifetime에 따라 static variable, stack (dynamic) variable, heap (dynamic) variable로 구분됨

```
x=95
x=x+5
print(x)
x="Seoul"
print(x+",Korea")
print(x+True) # type error
```

Dynamic typing → 실행 시점에 대입되는 값으로부터 변수 x에 타입 부여
Dynamic type checking → 실행 시점에 타입 검사



- 컴파일 시점(코드 int a;를 보고, 정적으로, static)에 변수 a에 타입 부여(typing, type binding) → Static typing
- b+1이 허용 가능 연산인지 컴파일 시점에 검사 → Static type checking

```
int a = 1 ;
int f() {
    int a=3, b=4;
    {
        int a;
        a=b+1;
        while(a--) b++;
    }
    return a+b;
}
int main() {
    int b=2;
    int v=(++a)*f()-b;
    return 0;
}
```

a의 scope
a, b의 scope
a의 scope
b, v의 scope

Scope (static scoping: 변수 사용 범위를 코드로부터 결정)

- 변수 a는 코드 전체에서 사용 가능 → global variable
- 변수 a는 a의 정의가 포함된 가장 가까운 함수에서만 사용 가능 → local variable
- 변수 a는 a가 정의된 블록에서만 사용 가능 → local variable

Lifetime

- 변수 a의 저장공간 lifetime은 프로그램 시작부터 종료까지 (a의 저장공간 주소는 불변) → static variable
- 변수 a의 저장공간 lifetime은 함수 f의 호출부터 종료까지 (a의 저장공간 주소는 함수 f 호출마다 다를 수 있음) → stack variable

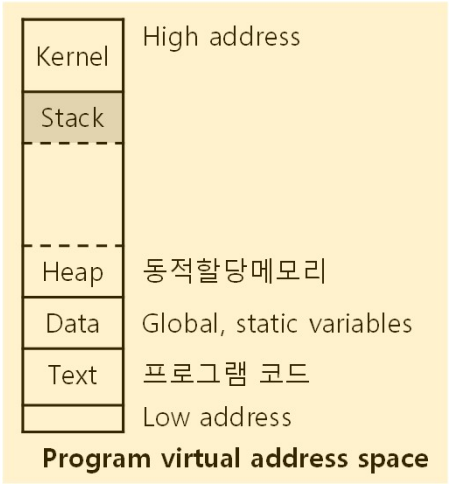
Variable: global/local 변수, static/stack 변수

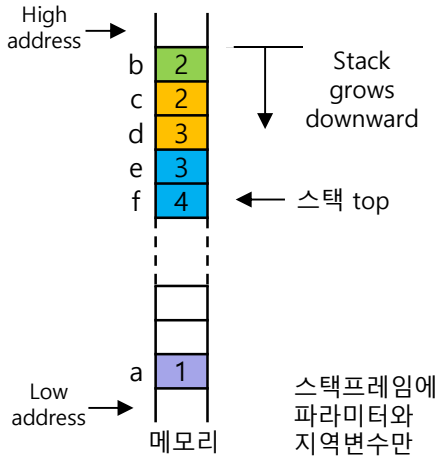
```
#include <stdio.h>
int a=1;
int g(int e){
    int f=4;
    return e-f+a;
}
int f(int c){
    int d=3;
    return c*g(d)+a;
}
int main() {
    int b=2;
    printf("%d", f(b));
    return 0;
}
```

local 변수

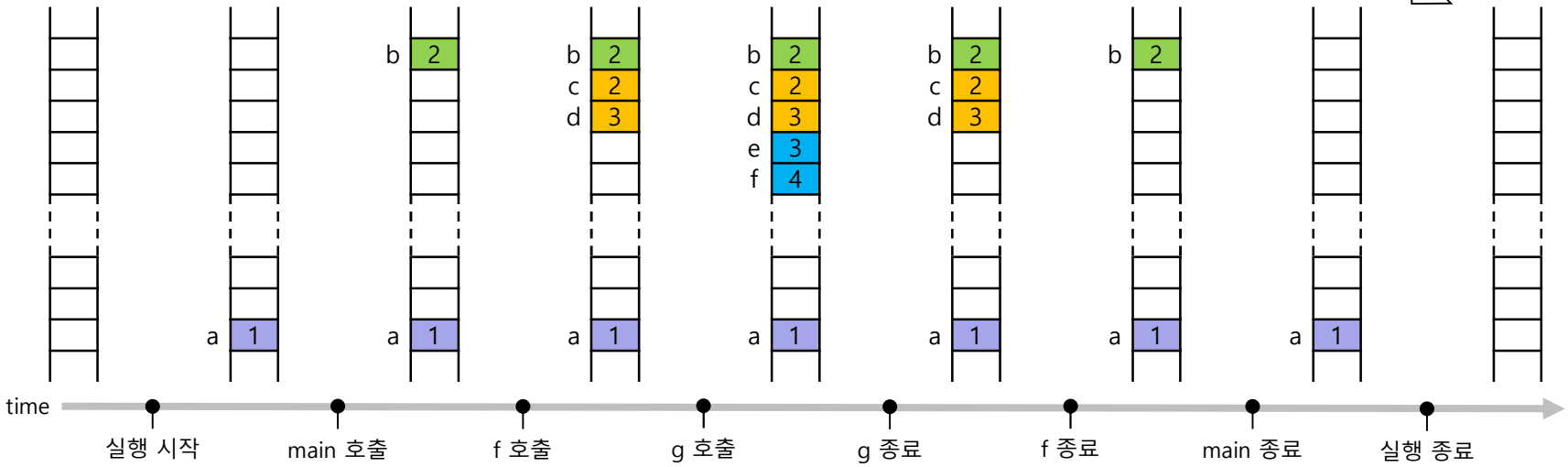
변수	Scope	Lifetime	
a	Global 변수	Static 변수	프로그램 실행 동안 존재
b	Local 변수	Stack 변수	main 실행 동안 stack에 존재
c	Local 변수	Stack 변수	f 실행 동안 stack에 존재
d	Local 변수	Stack 변수	f 실행 동안 stack에 존재

- 함수의 파라미터(parameter) 변수와 함수 내 지역변수는, 함수 호출(call) 시 스택에 저장공간을 부여 받아 존재하게 되고, 함수 종료(return) 시 사라짐



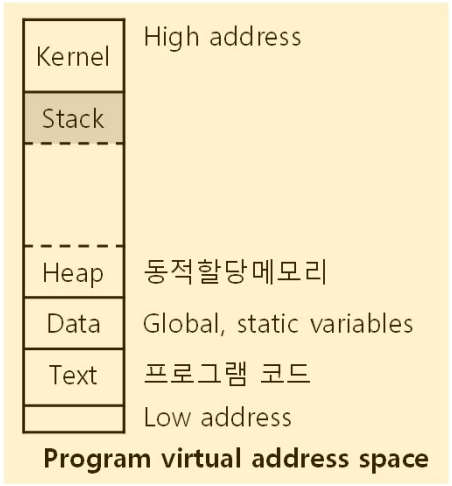


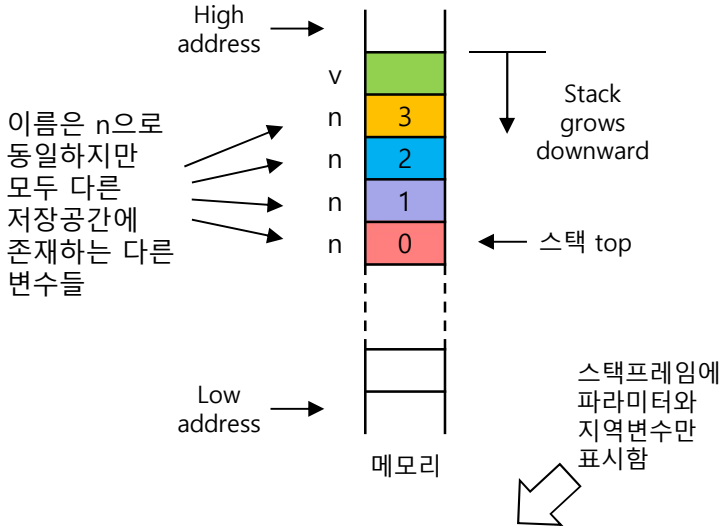
스택프레임에 파라미터와 지역변수만 표시함

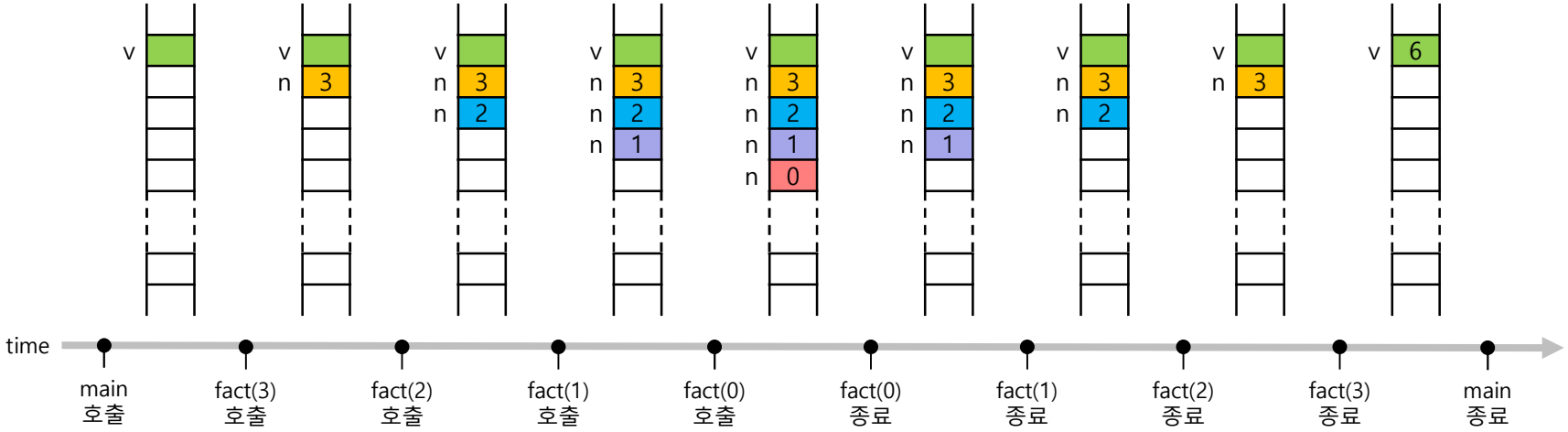


Variable: 재귀 호출과 스택 변수

```
#include <stdio.h>
int fact(int n){
    if(n==0) return 1;
    return n*fact(n-1);
}
int main(){
    int v=fact(3);
    printf("%d₩n",v);
    return 0;
}
```







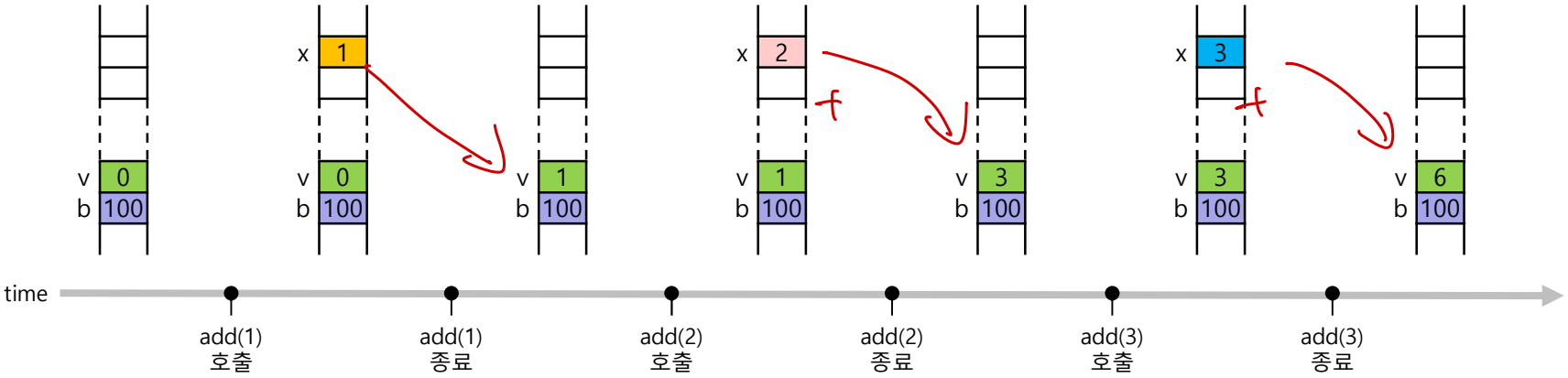
Variable: static local 변수

```
#include <stdio.h>
int b=100;
int add(int x){
   static int v=0;
   v=v+x;
   return b+v;
}
int main(){
   printf("%d₩n",add(1));
   printf("%d₩n",add(2));
   printf("%d₩n",add(3));
   return 0;
}
```

Static 이 붙어 있어 프로그램 종료시 사라짐

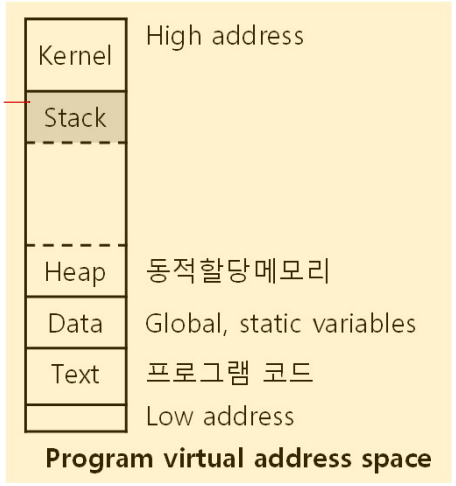
변수	Scope	Lifetime	
b	Global	Static	프로그램 실행 동안 존재
v	Local	Static	프로그램 실행 동안 존재
x	Local	Stack	add 실행 동안 stack에 존재

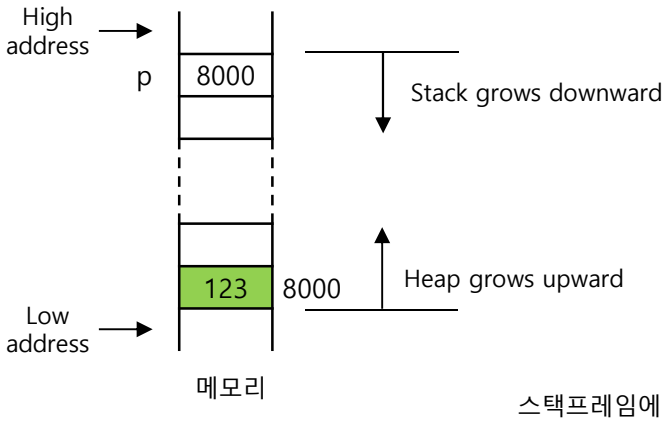
스택프레임에 파라미터와 지역변수만 표시함



Variable: heap 변수

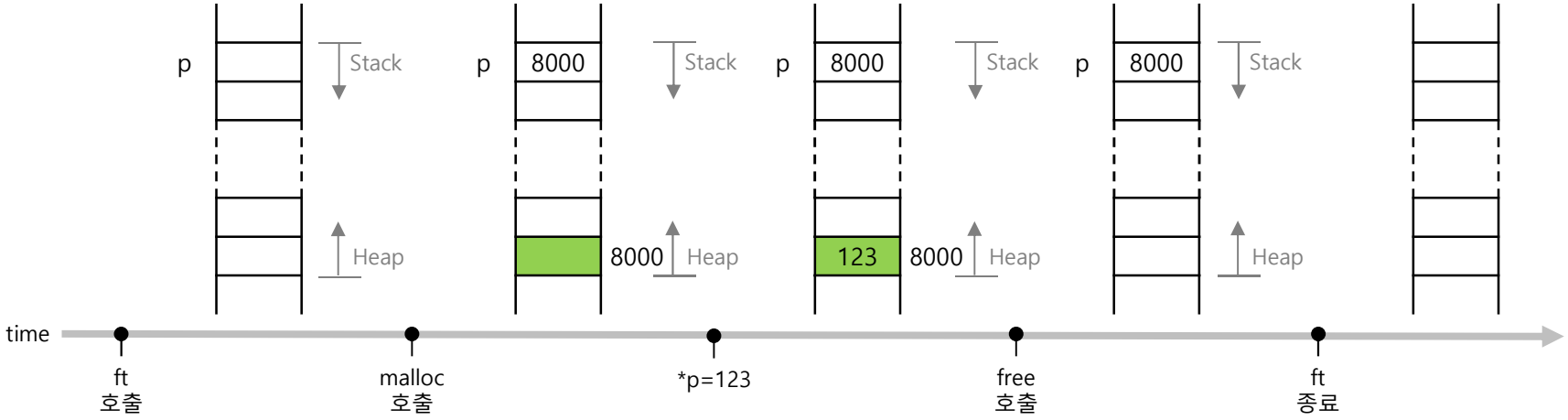
```
#include <stdio.h>
#include <stdlib.h>
void ft(){
    int *p;
    p=malloc(sizeof(int));
    *p=123;
    printf("%d₩n",*p);
    free(p);
}
int main(){
    ft();
    return 0;
}
```

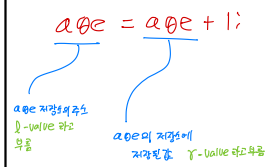




스택프레임에 파라미터와 지역변수만 표시함

C언어에서 heap 변수는 malloc을 통해 저장공간에 연결되어 사용되고 free를 통해 할당 받은 저장공간을 반환





Variable

Name(이름)

- Name은 프로그램 내 개체(변수, 부프로그램 등) 식별을 위해 사용되는 문자열로 식별자(identifier)라고도 함
- C 기반 언어 등 많은 PL에서 name의 대소문자 구별함(case sensitive)
- 대부분의 변수들은 이름을 가짐(이름 없는 변수도 있음)

Variable(변수)

- 변수는 메모리 셀(하나 혹은 모음)을 추상화한 것
- 변수는 6가지 속성(attribute)을 가짐 → **name, address, value, type, lifetime, scope**

Address(주소)

대입문의 왼쪽에 변수가 나타나면 l-value 라고 봄

- 변수의 address는 변수와 관련된 메모리 셀의 주소이다. 많은 PL에서 동일 변수는 다른 실행 시점에 다른 메모리 주소와 연결 가능(예: 런타임 스택에 할당되는 부프로그램 내 지역 변수의 주소는 부프로그램의 호출마다 다를 수 있음)
- 변수의 주소는 변수의 **l-value**라고도 함(대입문의 왼편에 출현한 변수에서 요구되는 것은 주소이기 때문)
- 동일 주소를 갖는 서로 다른 변수들을 **aliases**라고 부른다(실현 예: union 자료형, pointer 변수, 부프로그램 파라미터 등)
- 변수에 주소가 연결되는 시점은 매우 중요

Type(타입)

- 변수의 type은 변수에 저장 가능한 값의 범위와 연산을 결정한다 → 예) Java의 int 타입은 그 값의 범위가 -2147483648 ~ 2147483647이며 덧셈, 뺄셈, 곱셈, 나눗셈, 모듈로 연산을 명시한다

Value(값)

- 변수의 값은 변수와 연결된 메모리 셀(하나 혹은 모음)의 내용이다
- Java의 int 타입 변수의 값은 하나의 (추상적) 메모리 셀에 저장되지만 물리적으로는 4 바이트임
- 변수의 값은 **r-value**라고도 함(대입문의 오른편에 출현한 변수에서 요구되는 것은 값이므로)
- 변수의 **r-value**에 접근하기 위해서는 먼저 변수의 **l-value**가 결정되어야 함

Binding

Binding(바인딩)

- Binding이란 개체(entity, 예: 변수)와 속성(attribute)을 연결하는 것 → 예) 변수와 타입의 연결, 변수와 값의 연결, 연산과 심볼의 연결 등
- 바인딩이 발생하는 시점을 binding time이라 한다. 바인딩 타임의 예 → language design time, language implementation time, compile time, link time, load time, run time
- 바인딩 시점에 대한 이해는 프로그램의 의미 파악의 선행 요건
- 바인딩이 실행 시간 전에 발생하고 실행 동안 불변인 경우 그러한 바인딩은 **static**이라고 한다
- 바인딩이 실행 중에 처음 발생하거나 실행 동안 변경 가능한 경우 그러한 바인딩은 **dynamic**이라고 한다

바인딩 시점	예
Language design time	심볼 *를 곱셈 연산에 바인딩
Language implementation time	특정 데이터 타입이(예: C언어의 int 타입) 특정 값들의 범위에 바인딩
Compile time	프로그램 내 특정 변수가 특정 데이터 타입에 바인딩
Link time	라이브러리 내 특정 부프로그램 P에 대해, P의 호출을 P의 코드에 바인딩
Load time	프로그램 내 특정 변수는 프로그램 로드 시 메모리 셀에 바인딩
Run time	부프로그램 내 지역 변수는 부프로그램 호출 시 메모리 셀에 바인딩

전역변수

바인딩 시점은 변수를 선언하는 것도 포함하여 다 다르다.
물론 연산자마다 다 다르다

int count; **count = count + 5 ;**	count의 타입은 컴파일 타임에 바인딩 됨
	count의 가능한 값의 집합은 컴파일러 설계 시점에 바인딩 됨
	연산자 심볼 +의 의미는 컴파일 시점(즉, 피연산자의 타입이 결정될 때)에 바인딩 됨
	리터럴(literal) 5의 내부 표현은 컴파일러 설계 시점에 바인딩 됨
	count의 값은 이 문장 실행 시점에 바인딩 됨

예를들어 Int + Int, String + String 타입마다 다 다르다.

Type binding (1/2)

Type binding
- 변수는 사용 전에 데이터 타입에 바인딩되어 있어야 한다
- 변수에 타입을 명시하는 방법과 시점이 중요함
- Static type binding에서는 explicit declaration 혹은 implicit declaration을 통해 변수에 타입을 명시한다
- Explicit declaration은 변수 이름을 그 타입과 함께 명시하는 문장이다
- Implicit declaration은 변수 이름 최초 출현 시 변수에, 디폴트 관례에 따라 정해지는 타입을 연결하는 것으로, 변수 이름의 최초 출현이 implicit declaration이 된다
- Static type binding을 사용하는 대부분 PL에서는 모든 변수에 explicit declaration을 요구한다
- Implicit type binding은 컴파일러나 해석기에 의해 수행된다

Implicit (variable) type binding
- FORTRAN → 명시적으로 선언되지 않은 변수명의 첫 문자가 I, J, K, L, M, N, i, j, k, l, m, n 중 하나이면 해당 변수는 묵시적으로 Integer 타입이 되고 그렇지 않은 경우 묵시적으로 Real 타입이 된다
- Perl → $로 시작하는 변수는 scala이고(스트링이나 숫자 저장 가능), @로 시작하는 변수는 scala들의 array이고, %로 시작하는 변수는 hash이다(@score와 %score는 다른 변수임)
- C# → var age=23;과 같이 var로 시작하는 변수는 선언 시 반드시 초기화되어야 하는데 이 경우 (type inference를 통해 결정된) 초기값 23의 타입이 변수의 타입이 된다

Dynamic type binding
- Dynamic type binding에서는 변수의 타입이 선언문으로 명시되지 않으며 변수에 값 대입 시 대입되는 값의 타입으로 바인딩된다 → 변수에 어떤 타입이라도 바인딩 가능하며, 실행 중 몇 번이고 변수의 타입 변경 가능
- Python, JavaScript, PHP 등의 PL에서 타입 바인딩은 동적(dynamic)이다

Python
```
age=23
age=[23, 24, 21]
```

C#
```
dynamic age;
age=23;
age=23.5;
age="23";
```

Type binding (2/2)

```
// C:₩Windows₩Microsoft.NET₩Framework64₩v4.0.30319₩csc pgm1.cs
// 23
// Korea
using System;
namespace MyNameSpace {
    public class MyProgram {
        public static void Main(string []args){
            int age=23;
            double score=3.95;
            char code='A';
            bool isForeigner=false;
            string name="Gil-Dong Hong";

            //var age2; // error: 초기화 누락
            var age3=23; // age3는 int
            age3=3.14; // error: 암시적 변환(double->int) 오류
            Console.WriteLine(age3);

            dynamic v;
            v=23;
            v=3.95;
            v="Korea";
            Console.WriteLine(v);
        }
    }
}
```

Storage binding (1/2)

Storage binding
- 변수에 바인딩할 메모리 셀을 이용 가능 메모리 풀로부터 취하는 것을 allocation이라 하며 변수와 연결이 끊어진(unbound) 셀을 이용 가능 메모리 풀로 되돌리는 작업을 deallocation이라 한다
- 변수의 **lifetime**은 변수가 특정 메모리 셀에 바인딩되어 있는 동안의 시간이다
- 변수는 그 lifetime에 따라 4개 범주로 나눌 수 있다 → static, stack-dynamic, explicit heap-dynamic, implicit heap-dynamic

Static variable
- Static variable은 프로그램 실행 전에 특정 메모리 셀에 바인딩되고 프로그램 종료까지 동일 메모리 셀에 계속 바인딩되어 있는 변수이다 → 예) global variable(전역변수), local static variable
- Static 변수만 허용되는 언어는 재귀적 부프로그램 지원이 불가하다
- 둘 다 큰 배열들을 필요로 하는 부프로그램 A, B가 동시 실행되지 않는다고 할 때, 배열이 static이면 A, B는 배열을 위한 메모리 공간을 공유할 수 없다

Stack-dynamic variable
- Stack-dynamic variable은 변수 선언문이 실행될 때 실행 스택(run-time stack) 내 메모리 셀에 바인딩되는 변수이다
- 대부분 재귀적 부프로그램은 동적 지역 변수가 필요하며 이는 stack-dynamic variable로 구현 가능하다
- Stack-dynamic variable을 통해 부프로그램들은 같은 메모리 공간을 공유할 수 있다

Explicit heap-dynamic variable
- explicit heap-dynamic variable은 프로그래머가 작성한 명시적 명령에 의해 실행 시점에 heap에 할당(및 해제)되는 무명(nameless) 메모리 셀이며 포인터 변수나 참조 변수를 통해 접근 가능하다 → 예) C에서는 malloc(), free(), C++에서는 new, delete 연산자
- Java 객체는 explicit heap-dynamic이며 참조변수로 접근되지만, 할당 해제를 위한 명시적 방법은 없으며 이를 위해 garbage collection이 사용된다

Implicit heap-dynamic variable
- Implicit heap-dynamic variable은 실행 시 값이 대입될 때만 heap에 할당된 메모리 셀에 바인딩되는 변수이다
- Javascript 예 → v=[65,90,87];

Storage binding (2/2)

```
// gcc pgm.c
// a.exe
// 24
// 33
#include <stdio.h>
int fact(int n){
    if(n==0) return 1;
    return n*fact(n-1);
}
int sum(int x, int y){
    int z=x+y;
    return z;
}
int main(){
    printf("%d₩n",fact(4));
    printf("%d₩n",sum(11,22));
    return 0;
}
```

```
// gcc pgm.c
// a.exe
// 123
#include <stdio.h>
#include <stdlib.h>
int main(){
    int *p;
    p=malloc(sizeof(int));
    *p=123;
    printf("%d₩n",*p);
    free(p);
    return 0;
}
```

```
// g++ pgm.cpp
// a.exe
// 123
#include <stdio.h>
int main(){
    int *p;
    p=new int;
    *p=123;
    printf("%d₩n",*p);
    delete p;
    return 0;
}
```

```
// gcc pgm.c
// a.exe
// 101
// 103
// 106
// 99
// 97
// 94
#include <stdio.h>
int base=100;
int add(int x){
    static int v=0;
    v=v+x;
    return base+v;
}
int sub(int x){
    static int v=0;
    v=v-x;
    return base+v;
}
int main(){
    printf("%d₩n",add(1));
    printf("%d₩n",add(2));
    printf("%d₩n",add(3));
    printf("%d₩n",sub(1));
    printf("%d₩n",sub(2));
    printf("%d₩n",sub(3));
    return 0;
}
```

Scope, static scoping

Scope

- 변수가 문장(statement) 내에서 참조될 수 있으면 그 변수는 해당 문장에서 visible하다고 한다
- 변수의 scope이란 해당 변수가 visible한 문장(들)의 범위이다
- 특정 프로그램 단위나 블록(block)에서 선언된 변수를 local 변수라 한다
- 특정 프로그램 단위나 블록에서 선언되지 않았지만 visible한 변수를 nonlocal 변수라 한다 → 전역변수는 nonlocal 변수의 한 범주이다
- Static scoping(혹은 lexical scoping)은, nonlocal 변수에 name을 바인딩하는 방법으로 ALGOL 60에서 소개되었다
- Static scoping을 사용하는 PL은 부프로그램의 내포 여부에 따라 두 범주로 나뉜다 → Ada, JavaScript, Python 등은 부프로그램 내포를 허용하지만, C 기반 PL들은 그렇지 않다
- Static scoping에서는 이름에 대응하는 변수 선언을 찾기 위해 이름이 출현한 해당 부프로그램 S에서 먼저 찾고, 실패 시 S의 static parent(S를 선언한 부프로그램)에서 찾고, 실패 시 S의 static parent의 static parent에서 찾는 식으로 진행하여 최악의 경우 가장 큰 부프로그램 단위까지 진행한다

```
#include <stdio.h>
int x=1;
int f(){
    printf("%d₩n",x); // 1
}
int g(){
    int x=2;
    printf("%d₩n",x); // 2
}
int main(){
    f();
    g();
}
```

```
x=1
def f():
    print(x) # 1
# end def
def g():
    x=2
    print(x) # 2
# end def
f()
g()
```

```
x=1
def f():
    y=2
    def g():
        z=3
        print(x) # 1
        print(y) # 2
        print(z) # 3
    # end def
    g()
# end def
f()
```

```
x=1
def f():
    y=2
    def g():
        z=3
        y=-2
        print(x) # 1
        print(y) # -2
        print(z) # 3
    # end def
    g()
# end def
f()
```

- g()의 z에 대응하는 변수 선언을 찾기 위해, z가 출현한 부프로그램 g()에서 먼저 찾고, 실패 시 g()의 static parent인 f()에서 찾고, 실패 시 f()의 static parent에서 찾는다
- f()에 선언된 변수 y는 g()로부터 hidden된다

Block

Block(블록)

- 블록(block)은 문장(들)을 그룹으로 표현한 것으로 C 기반 언어의 블록은 {와 }로 감싸인 코드 부분으로 선언을 포함하며 새로운 scope을 정의한다
- 블록 내 변수는 stack-dynamic이며 블록의 시작과 끝에서 각각 저장공간의 할당과 해제가 발생된다
- 블록은 더 큰 블록에 내포될 수 있다
- 블록으로 만들어진 scope은 부프로그램으로 만들어진 scope처럼 다루어진다

```
#include <stdio.h>
int main(){
    int x=1;
    {
        int x=2;
        {
            int x=3;
            printf("%d₩n",x); // 3
        }
        printf("%d₩n",x); // 2
    }
    printf("%d₩n",x); // 1
}
```

```
#include <stdio.h>
int main(){
    int x=1;
    {
        int y=2;
    }
    printf("%d₩n",y); // error
}
```

```
#include <stdio.h>
int main(){
    int x=5;
    for(int i=0; i<=10; i++){
        if(i==x) break;
    }
    printf("%d₩n",i); // error
}
```

```
#include <stdio.h>
int main(){
    int x=1;
    while(x<=10){
        int prev=x;
        x=x+1;
    }
    printf("%d₩n",prev); // error
}
```

```
public class Test {
    public static void main(String[] args) {
        int x=5;
        if(x>0){
            int x; // 오류(duplicate local variable)
            x=100;
        }
    }
}
```

```
#include <stdio.h>
int main(){
    int x=5;
    if(x>0){
        int x;
        x=100;
    }
    printf("%d₩n",x); // 5
}
```

- if 문 블록 내 x는 if 문 블록 내 선언된 변수 x에 바인딩되며, main에서 선언된 변수 x는 if 문 블록으로부터 hidden된다
- Java와 C#의 경우 내포된 블록에서 (더 큰 블록에서 선언된) 같은 이름의 변수 선언 불가

```
#include <stdio.h>
int main(){
    int x=1;
    if(x>0){
        x=100;
        int x;
        x=-1;
        printf("%d₩n",x); // -1
    }
    printf("%d₩n",x); // 100
}
```

Global scope

Global scope

- C, C++, Python 등의 경우, 파일 내 함수 외부에 정의된 변수는 global이며 해당 파일 내 함수에서 visible 가능
- C, C++ → global 데이터에 대해 declaration과 definition을 둘 다 갖는다 → declaration은 타입 등 속성을 명시하지만 저장소는 할당하지 않으며, definition은 속성 명시와 함께 저장소도 할당 → C 프로그램은 특정 global 이름의 여러 선언을 포함할 수 있지만 정의는 하나만 포함
- C++ → 같은 이름의 local 변수에 의해 hidden된 전역변수 x는 ::x와 같이 scope 연산자 ::를 통해 접근 가능하다
- Python → global 변수 x는 함수 내에서 참조는 가능하지만, x=7과 같이 그 값을 변경하려면 global x로 선언되어 있어야 함 → global x를 통해 x가 전역변수라는 선언이 없다면 x=7 문장은 지역변수 x를 만듦

```
def f():
    print(x) # 3 (전역변수 x)

x=3
f()
print(x) # 3
```

```
def f():
    x=7
    print(x) # 7 (지역변수 x)

x=3
f()
print(x) # 3
```

```
def f():
    print(x)
    x=7
    print(x)

x=3
f()
print(x)
```

→ UnboundLocalError: local variable 'x' referenced before assignment

```
def f():
    global x
    print(x) # 3
    x=7
    print(x) # 7

x=3
f()
print(x) # 7
```

Dynamic scope

Dynamic scope

- Dynamic scoping은 부프로그램의 호출 순서에 의존하여 scope이 결정된다 → 즉 실행 시점에 결정됨
- Dynamic scoping에서는 이름에 대응하는 변수 선언을 찾기 위해 이름이 출현한 해당 부프로그램 S에서 먼저 찾고, 실패 시 S의 dynamic parent(S를 호출한 부프로그램)에서 찾고, 실패 시 S의 dynamic parent의 dynamic parent에서 찾는 식으로 진행하여 최악의 경우 가장 큰 부프로그램 단위까지 진행한다
- Dynamic scoping 단점 → nonlocal 변수에 대한 타입 검사를 정적으로 수행할 수 없다. 부프로그램 호출 순서를 알아야 비지역 변수의 의미 결정이 가능하므로 프로그램 가독성이 낮다. 비지역 변수로의 접근 시간이 정적 스코핑의 경우보다 더 오래 걸린다
- Dynamic scoping은 static scoping만큼 널리 사용되지 않는다

Dynamic scoping 가정
- sub2의 x는 sub1에 선언된 x임
- 호출순서: big→sub1→sub2

```
function big() {
   function sub1() {
      var x = 7;
      sub2();
   }
   function sub2() {
      var y = x;
      var z = 3;
   }
   var x = 3;
   sub1();
}
big();
```

Dynamic scoping 가정
- sub2의 x는 big에 선언된 x임
- 호출순서: big→sub2

```
function big() {
   function sub1() {
      var x = 7;
   }
   function sub2() {
      var y = x;
      var z = 3;
   }
   var x = 3;
   sub2();
}
big();
```

References

- 박두순. (2016). (내공 있는 프로그래머로 길러주는)컴파일러의 이해. 한빛아카데미.
- 창병모. (2021). 프로그래밍 언어론 : 원리와 실제. 인피니티북스.
- Aho, A., Lam, M., Sethi, R., Ullman, J. (2006). Compilers: Principles, Techniques, and Tools (2nd Ed.). Addison Wesley.
- Nystrom, R. Crafting interpreter. https://craftinginterpreters.com/
- Sebesta, R. (2012). Concepts of Programming Languages (10th. ed.). Pearson.
- Thain, D. (2023). Introduction to Compilers and Language Design.
- Paxson, V. (1995). Flex, version 2.5.
- Donnelly, C., Stallman, R. (2008). Bison.
- LexAndYacc.pdf (epaperpress.com)
- Tom Niemann, LEX & YACC. http://epaperpress.com/lexandyacc

References

- 박두순. (2016). (내공 있는 프로그래머로 길러주는)컴파일러의 이해. 한빛아카데미.
- 창병모. (2021). 프로그래밍 언어론 : 원리와 실제. 인피니티북스.
- Aho, A., Lam, M., Sethi, R., Ullman, J. (2006). Compilers: Principles, Techniques, and Tools (2nd Ed.). Addison Wesley.
- Nystrom, R. Crafting interpreter. https://craftinginterpreters.com/
- Sebesta, R. (2012). Concepts of Programming Languages (10th. ed.). Pearson.
- Thain, D. (2023). Introduction to Compilers and Language Design.
- Paxson, V. (1995). Flex, version 2.5.
- Donnelly, C., Stallman, R. (2008). Bison.
- LexAndYacc.pdf (epaperpress.com)
- Tom Niemann, LEX & YACC. http://epaperpress.com/lexandyacc