메모리 구조와 컴파일 과정

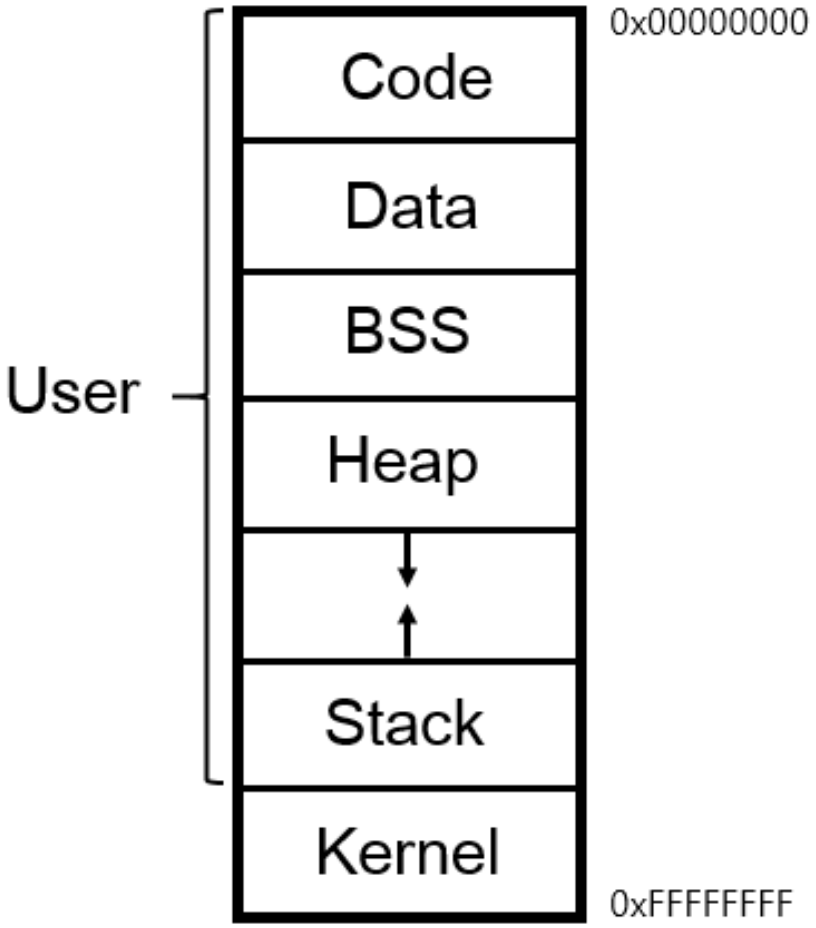
안태진, 이화경({taejin, hwakyung}@codecure.smuc.ac.kr)

상명대학교 보안동아리 Code Cure

목차

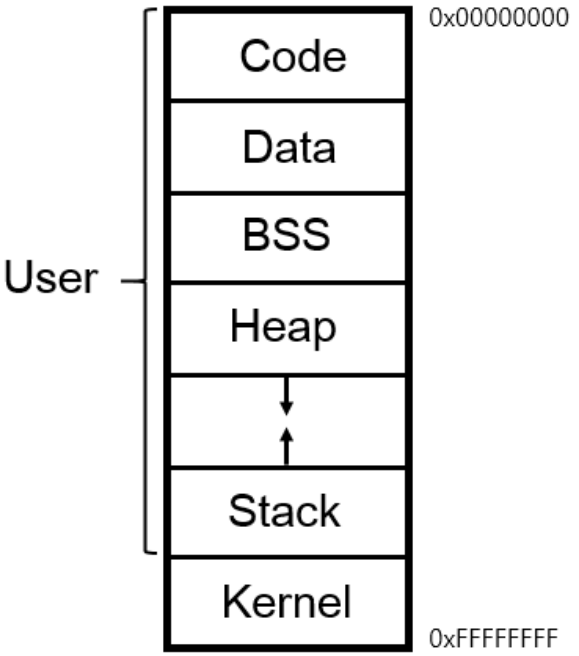
메모리 구조

- C


0x00000000
Code
Data
BSS
User
Heap
Stack
Kernel
0xFFFFFFFF

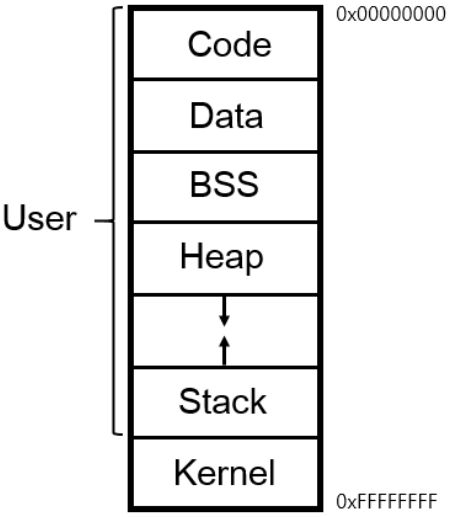
메모리 구조

- Kernel 영역(Kernel Space)
  - 운영체제의 핵심부분
    - 운영체제가 구동되는 영역
  - 시스템 운영에 필요한 메모리
    - 시스템 운영?
      - 메모리나 저장장치 내에서 주소공간 관리
  - 유저모드에서 접근불가능
    - 유저모드?
      - 유저 어플리케이션 코드만 실행 가능한 모드
      - 시스템 데이터, 하드웨어에 직접적인 접근이 불가능
    - 시스템 전체의 안전성과 보안을 지키기 위해 접근이 불가능함
      - 운영체제 데이터에 접근, 수정하지 못하게 함으로써 오동작을 유발하는 유저 애플리케이션이 시스템 전체의 안정성을 해치지 않게 함



메모리 구조

- User 영역(User Space)
  - 프로그램이 동작하기 위해 사용되는 메모리
  - Code, Data, BSS, Heap, Stack영역으로 나누어짐

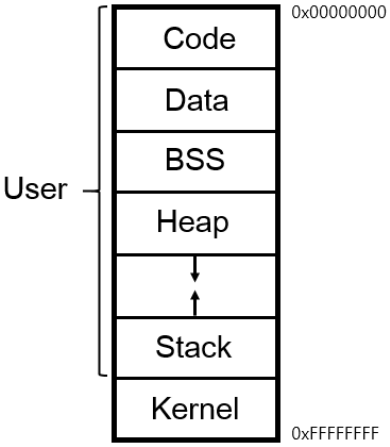


- Code 영역(Code Segment)
  - 작성한 소스코드 저장
  - 읽기 전용 데이터
  - CPU가 이 영역에 있는 명령을 읽고 처리
  - 컴파일 타임에 크기 결정, 크기가 변하지 않음
    - 컴파일 타임?
      - 프로그래밍 언어를 기계어로 변경하는 과정

메모리 구조

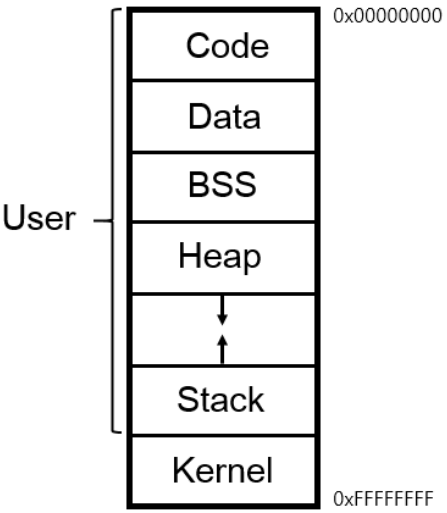
- Data 영역(Data Segment)
  - 프로그램 시작과 동시에 할당되고 프로그램 종료 시 소멸
  - 초기화 된 전역변수, static변수
  - 컴파일 타임에 크기 결정, 크기가 변하지 않음

- BSS(Block Stated Symbol) 영역(BSS Segment)
  - 초기화 되지 않은 전역변수, Static변수
  - 컴파일 타임에 크기 결정, 크기가 변하지 않음
  - 데이터 영역과 BSS 영역을 나누는 이유
    - 실행파일(프로그램) 크기를 줄이기 위해서임



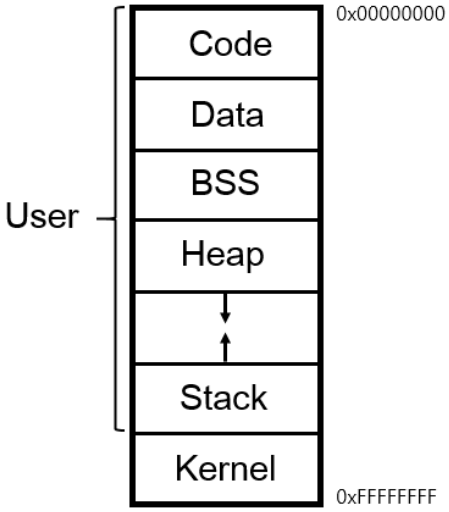
메모리 구조

- Heap 영역(Heap Segment)
  - 동적 할당
  - 사용자가 직접 관리할 수 있는 메모리 영역
  - 낮은 주소에서 높은 주소로 할당
  - 런타임에 크기 결정
    - 런타임?
      - 컴파일이 끝나 생성된 실행파일(프로그램)이 실행되고 있는 때, 컴퓨터 내에서 프로그램이 기동되고 있을 때



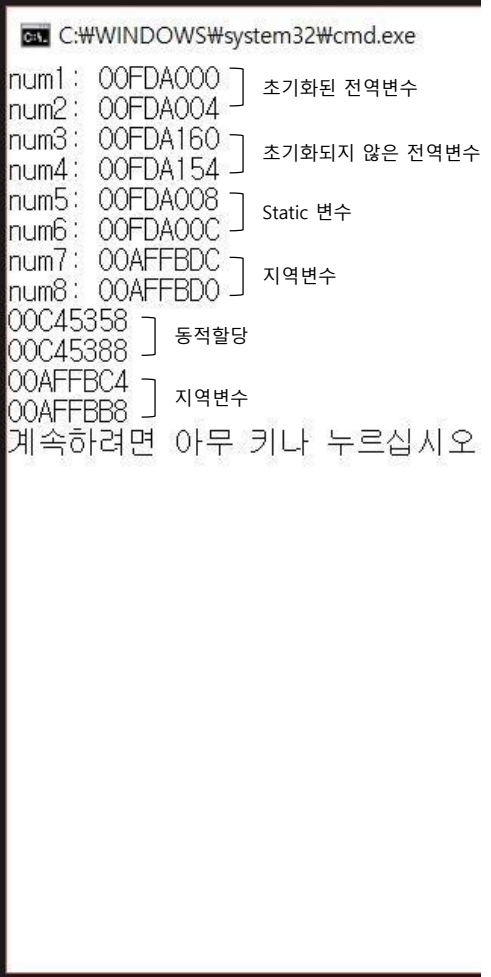
메모리 구조

- Stack 영역(Stack Segment)
  - 지역변수, 매개변수, return값 등
  - 높은 주소에서 낮은 주소로 할당
  - 컴파일 타임에 크기 결정
  - LIFO(Last In First Out)
    - 후입선출, 나중에 들어온 것이 먼저 나감
  - push동작으로 데이터 저장, pop동작으로 데이터 인출
    - push동작, pop동작?
      - 스택에 데이터를 추가하거나 빼내는 명령어



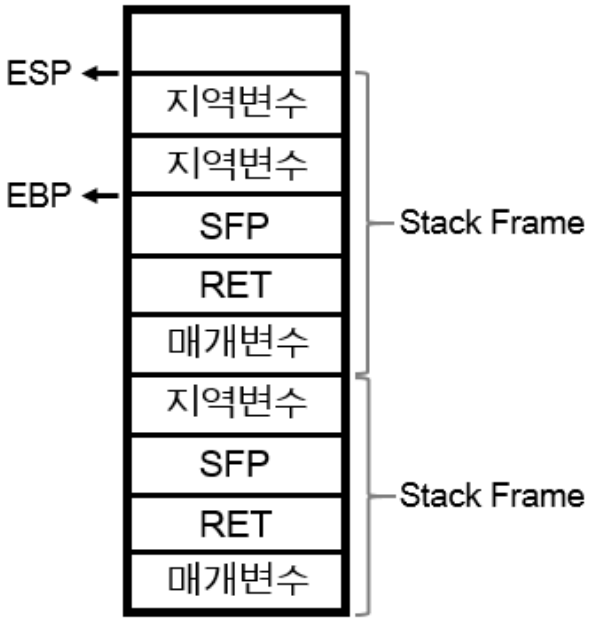
메모리 구조

```c
int num1 = 10;
int num2 = 20;
int num3;
int num4;

int main(void) {
    static int num5 = 50;
    static int num6 = 60;
    int num7 = 70;
    int num8 = 80;

    int* numPtr1 = malloc(sizeof(int));
    int* numPtr2 = malloc(sizeof(int));

    printf("num1: %p\n", &num1);
    printf("num2: %p\n", &num2);
    printf("num3: %p\n", &num3);
    printf("num4: %p\n", &num4);
    printf("num5: %p\n", &num5);
    printf("num6: %p\n", &num6);
    printf("num7: %p\n", &num7);
    printf("num8: %p\n", &num8);
    printf("%p\n", numPtr1);
    printf("%p\n", numPtr2);
    printf("%p\n", &numPtr1);
    printf("%p\n", &numPtr2);
```



메모리 구조

- 스택 프레임(Stack Frame)
  - 함수 호출 시 사용되는 스택 공간
  - ESP(Extend Stack Pointer)
    - 스택 프레임의 가장 최상부, 현재위치
  - EBP(Extend Base Pointer)
    - 스택프레임이 시작된 주소
  - RET
    - 복귀할 함수의 주소
  - SFP(Saved Frame Pointer)
    - EBP 복귀 주소 저장

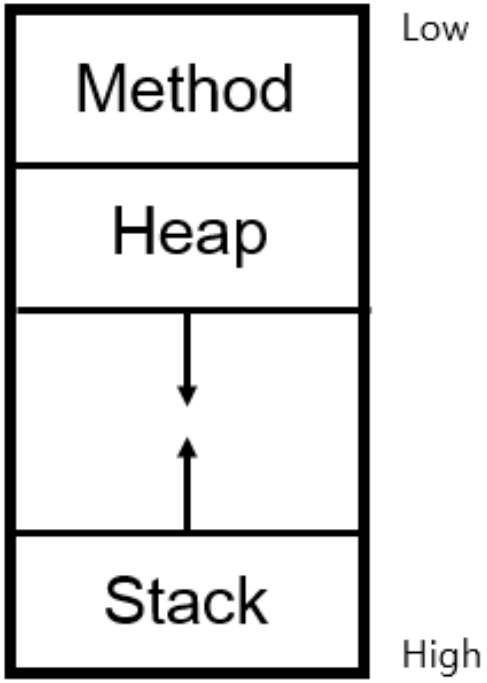


목차

- 메모리 구조
  - C
  - JAVA
- 컴파일 과정

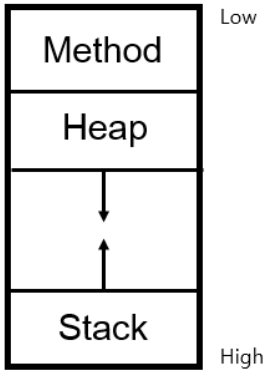
메모리 구조

- JAVA
  - 자바 가상 머신(JVM)을 통해 프로그램 실행
  - 운영체제에게 메모리 공간을 할당 받음



메모리 구조

- Method 영역
  - 클래스(Class) 정보, 클래스 변수(Static Variable)
    - 클래스(Class)?
      - 객체를 정의하는 틀, 설계도

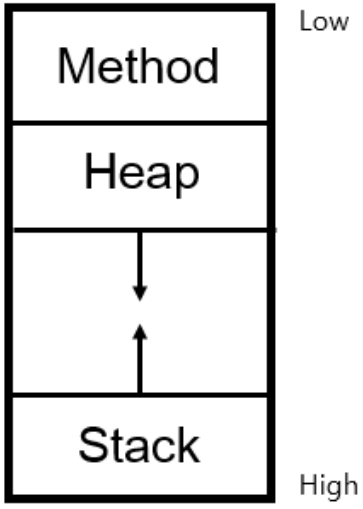


메모리 구조

- Heap 영역
  - 인스턴스(Instance) 변수
    - 선언된 클래스 타입의 객체
  - 참조하는 변수가 없다면 쓰레기 수집기(Garbage Collector)를 실행시켜 자동으로 제거
    - 쓰레기 수집기(Garbage Collector)?
      - 유효하지 않는 주소를 가진 메모리를 Garbage라고 함
        - 유효하지 않은 주소?
          - 모든 객체 참조가 null인 경우
          - 객체가 블록 안에서 생성되고 블록이 종료된 경우
        - 동적으로 할당했던 메모리 영역 중 Garbage를 찾아내고, Garbage 객체를 반환하여 메모리 회수

메모리 구조

- Stack 영역
  - 지역변수, 매개변수
  - 메소드(Method) 호출과 동시에 할당, 완료 시 소멸
    - 메소드(Method)
      - 객체의 행동, 특정 작업을 수행하기 위한 명령문의 집합

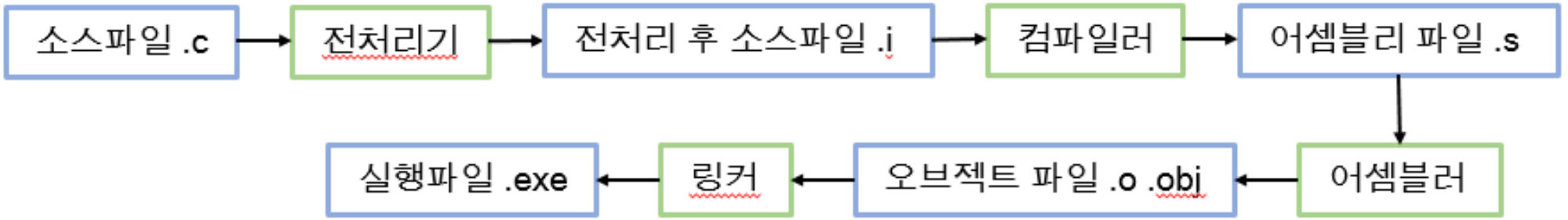


목차

컴파일 과정

- 컴파일(Compile)
  - 고급언어를 저급언어로 번역하는 것
    - 고급언어?
      - 인간이 구분하기 쉬운 프로그램 언어, 소스코드
    - 저급언어?
      - 컴퓨터 측면에서 사용하기 쉬운 언어
      - e.g., 어셈블리어, 기계어(이진코드)
        - 어셈블리어(assembly language)?
          - 니모닉 기호(mnemonic symbol)로 표현한 언어
            - 일대일 대응으로 이진코드로 이루어진 명령어를 사람이 알아보기 쉬운 명령어로 표현한 기호

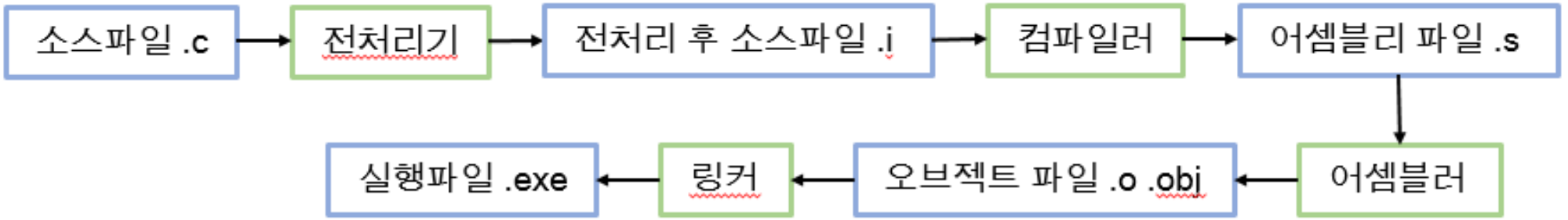
컴파일 과정



1. 전처리기(Preprocessor)

- 컴파일 전 헤더파일 삽입, 매크로 치환, 주석제거 등 수행
  - 매크로 치환?
    - 반복적으로 나타나는 함수나 상수를 새롭게 정의한 것을 바꾸는 작업
- .i 파일 생성

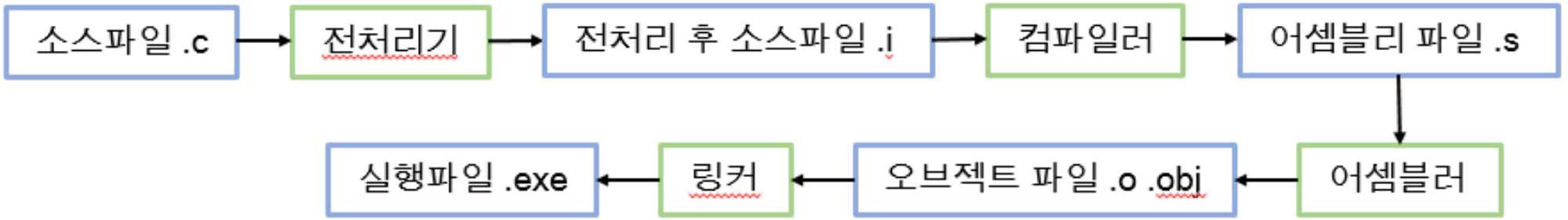
컴파일 과정



2. 컴파일러(Compiler)

- 고급언어를 저급언어로 번역해주는 프로그램
- 어셈블리 파일 생성
  - 어셈블리 파일?
    - 어셈블리어로 작성된 파일

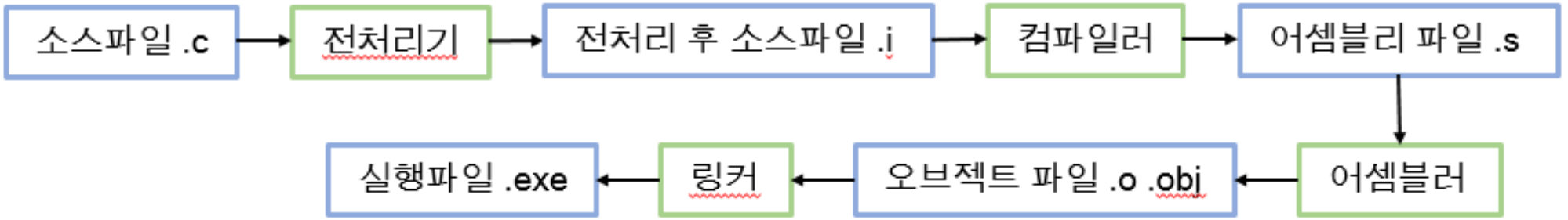
컴파일 과정



3. 어셈블러(Assembler)

- 어셈블리 언어를 기계어로 번역해주는 프로그램
- 오브젝트 파일 생성
  - 오브젝트(Object) 파일?
    - 이진코드로 작성된 파일

컴파일 과정



4. 링커(Linker)

- 여러 개의 오브젝트 파일을 하나로 연결해 실행파일 생성
  - 동적링킹(Dynamic Linking)
    - 프로그램 실행시 라이브러리를 읽어들이는 방법
  - 정적링킹(Static Linking)
    - 라이브러리를 실행 프로그램에 같이 링킹 하는 방법

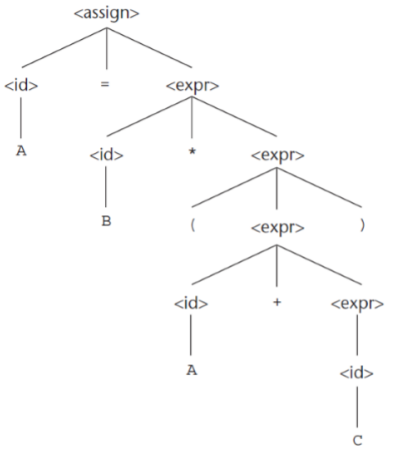
컴파일 과정

- 컴파일러(Compiler) 처리 과정
  1. 어휘 분석
     - 소스코드를 토큰(Token)단위로 나눔
       - 토큰(Token)?
         - 의미가 있는 가장 작은 단위
         - e.g., +, =, 5

컴파일 과정

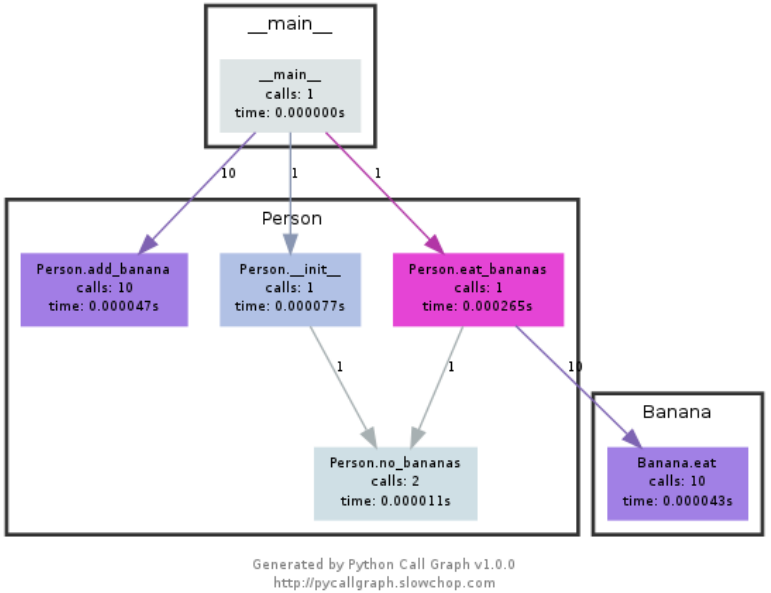
2. 구문 분석
   - 문법적 오류 검사
   - 파스 트리(Parse Tree) 생성
     - 파스 트리?
       - 문장에 대한 계층적 구조 표현
     - 장점
       - 전체 식에 대한 평가 순서 파악 용이
     - 한계
       - 모호성이 생길 수 있음
         - 모호성?
           - 주어진 문장이 2개 이상의 다른 파스 트리를 갖는 문장 형태

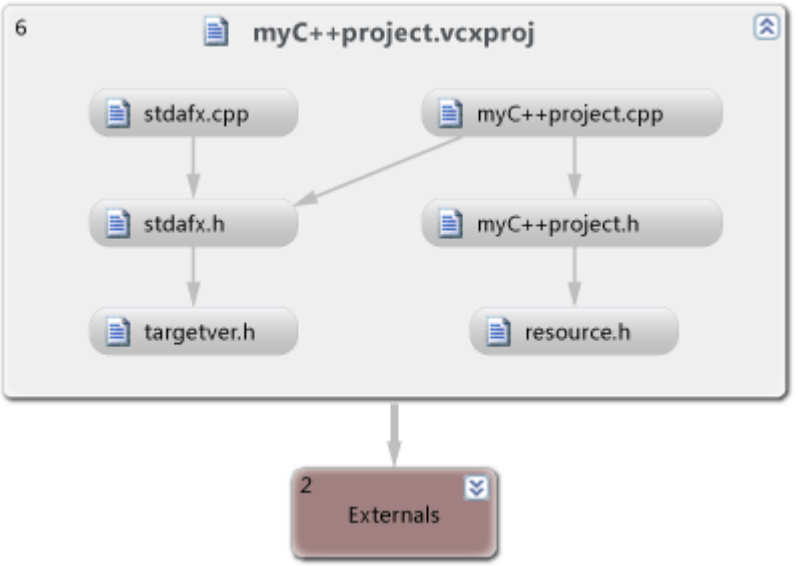
<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <id> + <expr>
| <id> * <expr>
| (<expr>)
| <id>



컴파일 과정

- 비슷한 표현법
  - 추상 구문 트리(Abstract Syntax Tree, AST)
    - 파스 트리를 간략히 표현
    - 트리에 괄호나 세미콜론(;)등이 사라짐
  - 콜 그래프(Call Graph)
    - 프로그램의 함수들 사이의 호출 관계를 표현한 그래프
  - 종속성 그래프(Dependency Graph)
    - 여러 프로젝트, 라이브러리, 객체들의 종속성을 나타내는 그래프





컴파일 과정

3. 의미 분석
   - 의미상 오류 검사
     - e.g., 타입 오류 검사
4. 중간코드 생성
   - 어셈블리어에 가까운 코드
   - 중간 언어? (Intermediate Language, IL)
     - 중간 코드를 생성하기 위한 언어
   - 중간 표현? (Intermediate Represent)
     - 중간 코드를 생성하기 위해 만드는 표현
     - 어휘 분석과 구문 분석을 거쳐 만들어진 표현
5. 최적화
   - 코드 크기는 줄이고 실행 속도를 높이는 작업
6. 어셈블리 파일 생성

감사합니다!