1. **일반선언문(declaration)은 어떻게 생겼는지 예를 들어가며 설명하시오.**

C언어에서의 일반선언문은 다음과 같이 정의 할 수 있다.

|  |
| --- |
| declaration // 선언문  : declaration\_specifiers init\_declarator\_list; // 타입정의문 변수선언  declaration\_specifiers:  : type\_specifier (struct, enum, int, double …)  | storage\_class\_specifier (static, register …)  | type\_qualifier  | type\_specifier declaration\_specifiers  | storage\_class\_specifier declaration\_specifiers  | type\_qualifier declaration\_specifiers  storage\_class\_specifier  : auto  | static  | typedef  | register  | extern  type\_qualifier  : const  | volatile  init\_declaration\_list (a | a, b | …)  : init\_declaration  | init\_declaratior\_list, init\_declarator  init\_declarator  : declarator  | declarator = initializer |

기본적으로 일반선언문은 변수선언을 예로 들 수 있다. 정수형타입 변수 a가 10이라는 것을 선언하기 위해서 우리는 c언어에서는 int a = 10;아라고 작성하게 된다. 이때 int는 declaration\_specifiers에 포함되고, a=10은 init\_declaration\_list에 포함된다. List라는 단어 자체의 의미로도 복수의 개수를 포함하고 있듯, 하나 혹은 여러 개를 사용할 수 있다. declaration\_specifiers는 여러 개로 다시 구분지어진다. 앞서 제시한 예시에서 int는 type\_specifier임을 알 수 있다. 이러한 declaration\_specifiers 중 type\_specifier에는 struct, enum, type\_identifier가 있다. 또한 static, register 같은 storage\_class\_specifier도 될 수 있다. 위의 일반선언문의 각 부분은 다음과 같이 예시를 들 수 있다.

|  |
| --- |
| static int x = 10;  typedef struct \_name {  char first\_name[10];  char last\_name[10];  };  // type\_specifier: int, struct, char  // init\_declarator: x, first\_name, last\_name, \_name  // init\_declarator\_list: x, first\_name, last\_name, \_name  // storage\_class\_specifier: typedef, static |

1. 수식 (expression)은 어떻게 생겼는지 예를 들어가며 설명하시오.

먼저 기본적으로 수식문은 다음과 같이 표현된다.

|  |
| --- |
| expression\_statement  : ;  | expression ; |

즉 세미콜론을 기준으로 수식문이 구분지어진다는 것을 알 수 있다. 그리고 이 expression은 수식과 연산자로 이루어져 있다. 기본적으로 수식과 연산자는 정해진 우선순위에 따라 문법이 순차적으로 구성되어 있다. 첫 번째로 우선순위가 가장 높은 primary\_expression, 그리고 postfix\_expression, arg\_expression\_list, unary\_expression, cast\_expression, multiplicative\_expression, additive\_expression, shift\_expression, relational\_expression, equality\_expression, logical\_expression, conditional\_expression, assignment\_expression, comma\_expression, constant\_expression가 존재한다. 각 수식문의 내용은 다음과 같이 표현될 수 있다.

|  |
| --- |
| primary\_expression  : IDENTIFIER  | INTEGER\_CONSTANT // int  | FLOAT\_CONSTANT // float  | CHARACTER\_CONSTANT // char  | STRING\_LITERAL // string  | ( expression ) // ( )  postfix\_expression  : primary\_expression  | postfix\_expression [ expression ] // arr[]  | postfix\_expression ( expression\_listopt ) // func()  | postfix\_expression . IDENTIFIER // id.local  | postfix\_expression -> IDENTIFIER //  | postfix\_expression ++ // a ++  | postfix\_expression – // a --  arg\_expression\_list  : assignment\_expression  | expression\_list , assignment\_expression // a, b  unary\_expression  : postfix\_expression  | ++ unary\_expression // ++a  | -- unary\_expression // --a  | & cast\_expression // &a  | \* cast\_expression // \*a  | ! cast\_expression // !a  | - cast\_expression // -a  | + cast\_expression // +a  | sizeof unary\_expression // sizeof(a)  | sizeof ( type\_name ) // sizeof(int)  cast\_expression  : unary\_expression  | ( type\_name ) cast\_expression // (int)a  type\_name  : declaration\_specifiers  | declaration\_specifiers abstract\_declarator // int [](추상화)  multiplicative\_expression  : cast\_expression  | multiplicative\_expression \* cast\_expression // a \* b  | multiplicative\_expression / cast\_expression // a / b  | multiplicative\_expression % cast\_expression // a % b  additive\_expression  : multiplicative\_expression  | additive\_expression + multiplicative\_expression // a + b  | additive\_expression - multiplicative\_expression // a - b  shift\_expression  : additive\_expression  | shift\_expression << additive \_expression // a << 2  | shift\_expression >> additive \_expression // b >> 2  relational\_expression  : shift\_expression  | relational\_expression < shift \_expression // a < b  | relational\_expression > shift \_expression // a > b  | relational\_expression <= shift \_expression // a <= b  | relational\_expression >= shift \_expression // a >= b  equality\_expression  : relational\_expression  | equality\_expression == relational\_expression // a == b  | equality\_expression != relational\_expression // a != b  logical\_and\_expression // a && b  : equality\_expression  | logical\_and\_expression && equality\_expression  logical\_or\_expression // a || b  : logical\_and\_expression  | logical\_or\_expression || logical\_and\_expression  conditional\_expression // a == b ? true : false  : logical\_or\_expression  | logical\_or\_expression ? expression : conditional\_expression  assignment\_expression // a = b  : conditional\_expression  | unary\_expression = assignemnt\_expression  comma\_expression // a, b  : assignment\_expression  | comma\_expression , assignment\_expression  expression  : comma\_expression  constant\_expression // 상수  : expression |

1. 여태까지 C의 문법과 관련하여 잘못 알고있었거나 모르고 있었던 사항이 있으면 모두 설명하시오.

지금까지 대학에 진학 후 코드를 이해하고 학습을 하면서 당연하다고 생각했던 규칙들이 상세한 규칙과 함께 정의가 되어있다는 것이 새로웠다. 특히 문법론적 부분이 맞으면 의미로도 당연히 맞다고 생각했다. 그리고 컴퓨터는 문법적인 측면을 더 잘 파악할 것이라 예상했고, 나 스스로가 작성하는 코드도 의미론적인 부분보다는 문법적인 측면을 더 따르고 있을 것이라 생각했다. 그러나 float int a, b = 0과 같은 선언의 예시로 보면서 실제 코드에서는 문법적인 측면보다 의미적인 측면을 더 중요시하게 작성을 하고 있었음을 새로이 알 수 있었다. 그리고 작년에 스택 계산기 코드를 작성하면서 수식 기호에 우선순위가 존재한다는 점을 기억하고 있었는데, 코드 상에서도 이러한 수식 기호와 더불어서 선언문의 표현식에서도 각 부표현식마다의 우선순위가 다름을 처음으로 알게 되었다. 그리고 재귀를 공부하면서 함수의 호출 순서와 마찬가지로 선언문에서의 표현식이 도출되는 것도 비슷한 매커니즘을 따라가고 있다는 것을 알 수 있었다.