컴파일러 과제 4

중간고사 문제풀기

제출마감일자 : 2016.11.14

담당교수 : 유재우 교수님
소속학부 : 컴퓨터학부
학번 : 20142577
이름 : 홍상원

(1) Recursive-descent parser 가 올바로 작동하기 위한 syntax diagram 의 조건 혹은 형태에 관하여 아는대로 작성하시오.

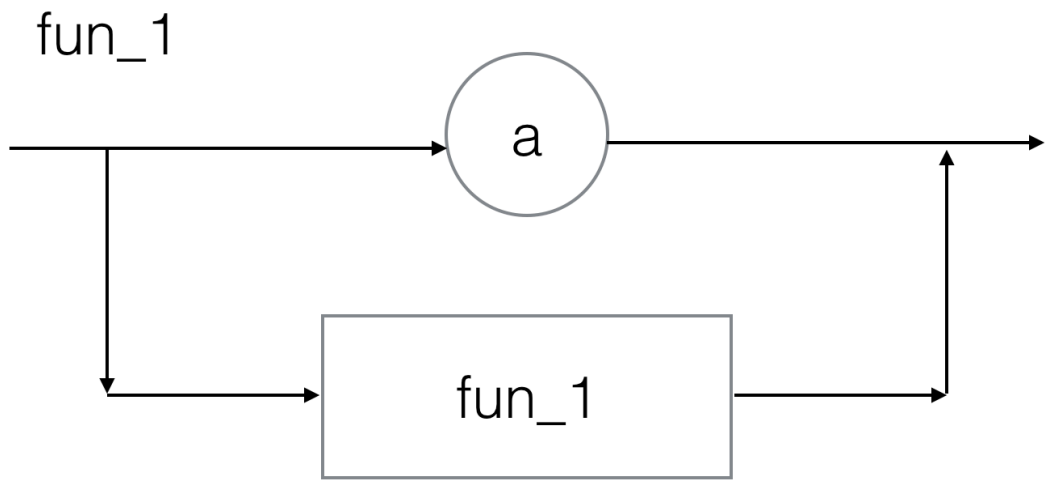
Recursive-descent parser(재귀 하강식 파서)란 재귀적 방식으로 작동하는 하향식(top-down) 파서를 의미한다.

Context-free Grammar G = (T, N, P, S)
P:
〈expression〉 ::= 〈term〉 〈expr_rest〉
〈expr_rest〉 ::= + 〈term〉 〈expr_rest〉
| λ
〈term〉 ::= 〈factor〉 〈term_rest〉
〈term_rest〉 ::= * 〈factor〉 〈term_rest〉
| λ
〈factor〉 ::= 〈number〉
| (〈expression〉)
〈number〉 ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

위의 Context-free Grammar G 의 BNF 표기법을 통해 알 수 있듯이 Recursive-descent parser 는 〈expression〉에서 시작하여 하향식 방식으로 〈term〉, 〈factor〉를 차례로 호출한다.

Recursive-descent parser 가 올바르게 작동하기 위해서는 조건(제한사항)이 존재한다. 다음에 제시된 경우 1, 2, 3, 4 는 Recursive-descent parser 가 올바르게 작동하기 위해 경계해야 할 syntax diagram 설계 방식을 보여준다.

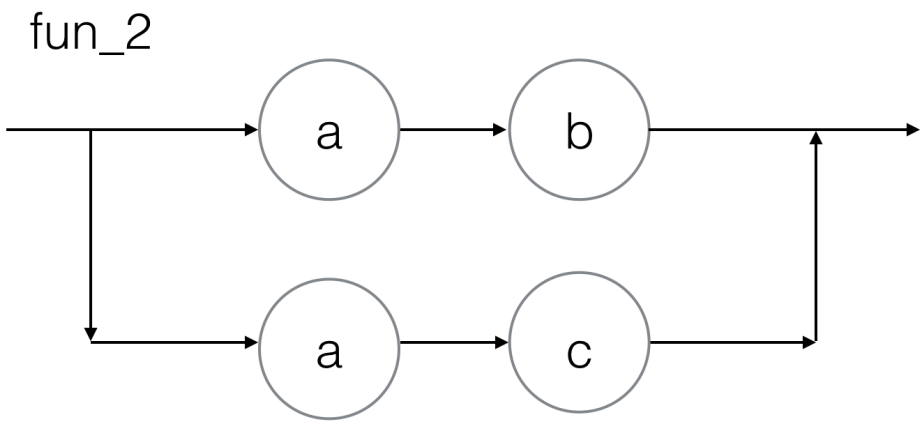
경우 1



```
void fun_1() {
   if(token=='a')
      get_token();
   else
      fun_1();
}
```

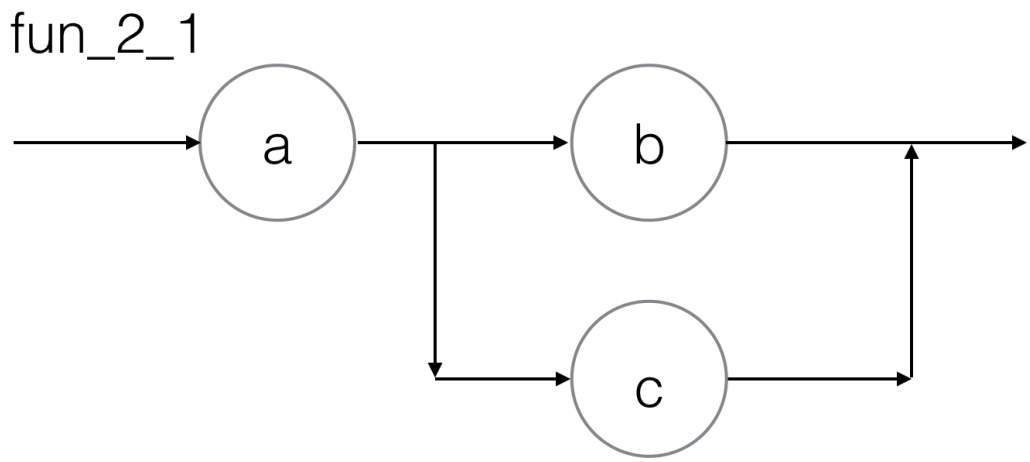
경우 1 의 의도는 parser 가 여러 개의 a 로 구성된 문자열을 인식하는 것이다. 그러나 a 이외의 다른 토큰을 입력받으면 그 토큰을 인식할 수 없고 fun_1 함수의 재귀적 호출에 의해 무한 루프에 빠진다.

경우 2




```
void fun_2() {
   if(token == 'a') {
      get_token();
      if(token == 'b')
         get_token();
      else
         error();
   }
   else if(token == 'a') {
      get_token();
      if(token == 'c')
         get_token();
      else
         error();
   }
   else error();
}
```

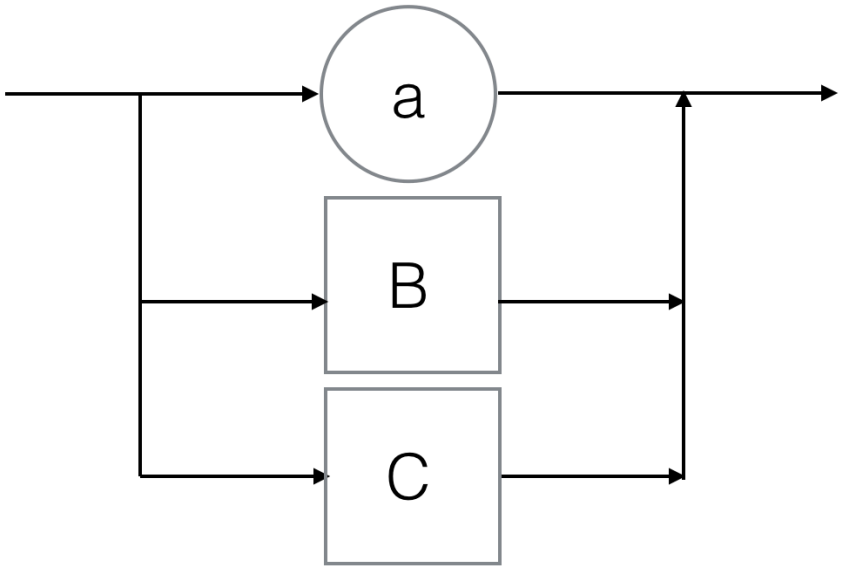
경우 2 의 의도는 parser 가 ab 또는 ac 문자열 패턴을 인식하는 것이다. 그러나 처음에 a 가 입력된 다음 b 가 입력되지 않으면 오류로 처리한다. 따라서 만약 ac 를 입력한다면 parser 는 ac 를 오류로 처리된다. 위와 같이 두 패턴 중 처음 토큰이 일치할 경우에는 처음 동일한 토큰을 인식한 후에 그 다음 토근의 조건에 따라 나누는 것이 바람직하다.



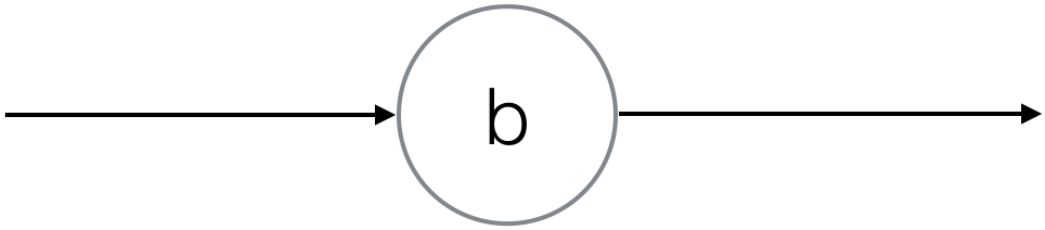

```
void fun_2_1() {
   if(token == 'a') {
      get_token();
      if(token == 'b')
         get_token();
      else if(token == 'c')
         get_token();
      else
         error();
   }
   else error();
}
```

경우 3

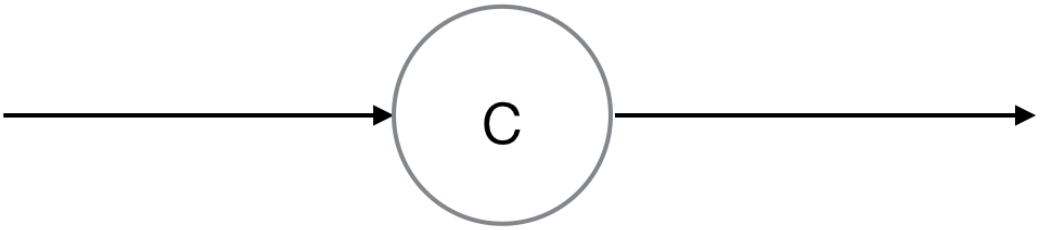
fun_3



B



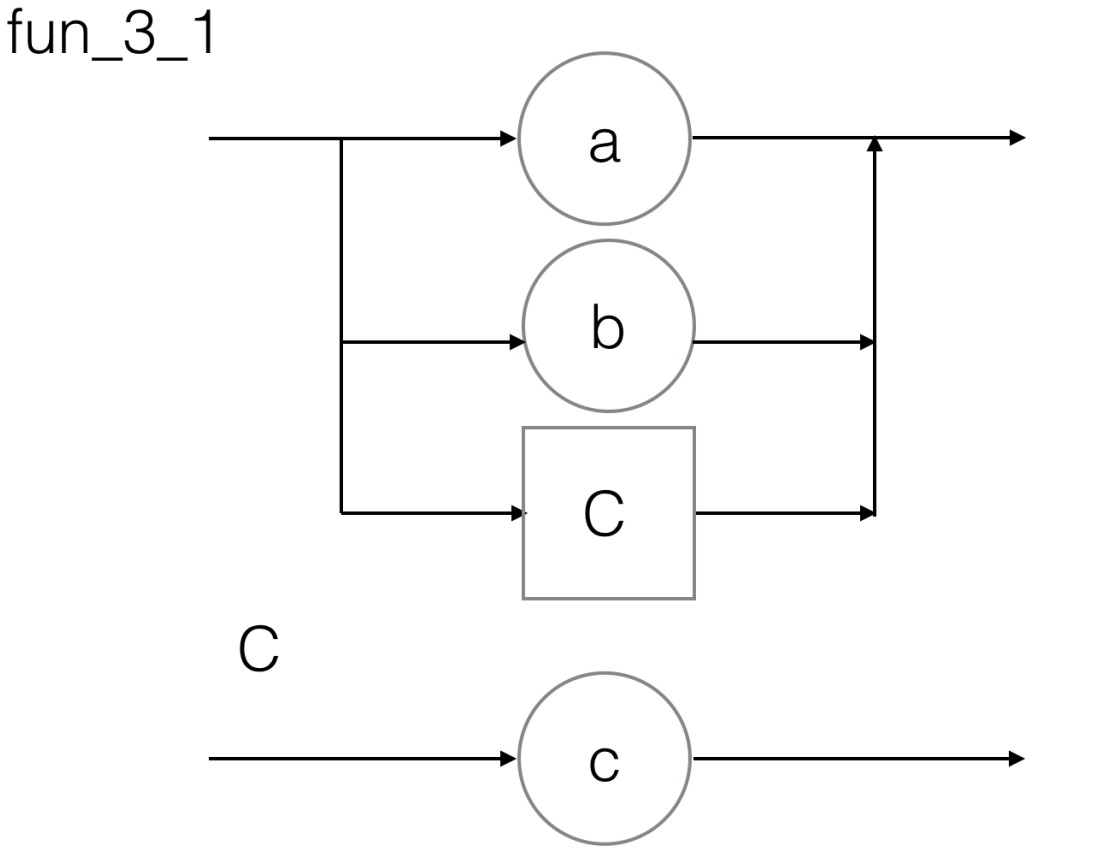
C




```
void fun_3() {
    if(token == 'a')
        get_token();
    else
        B();
    else // 오류
        C();
}
```

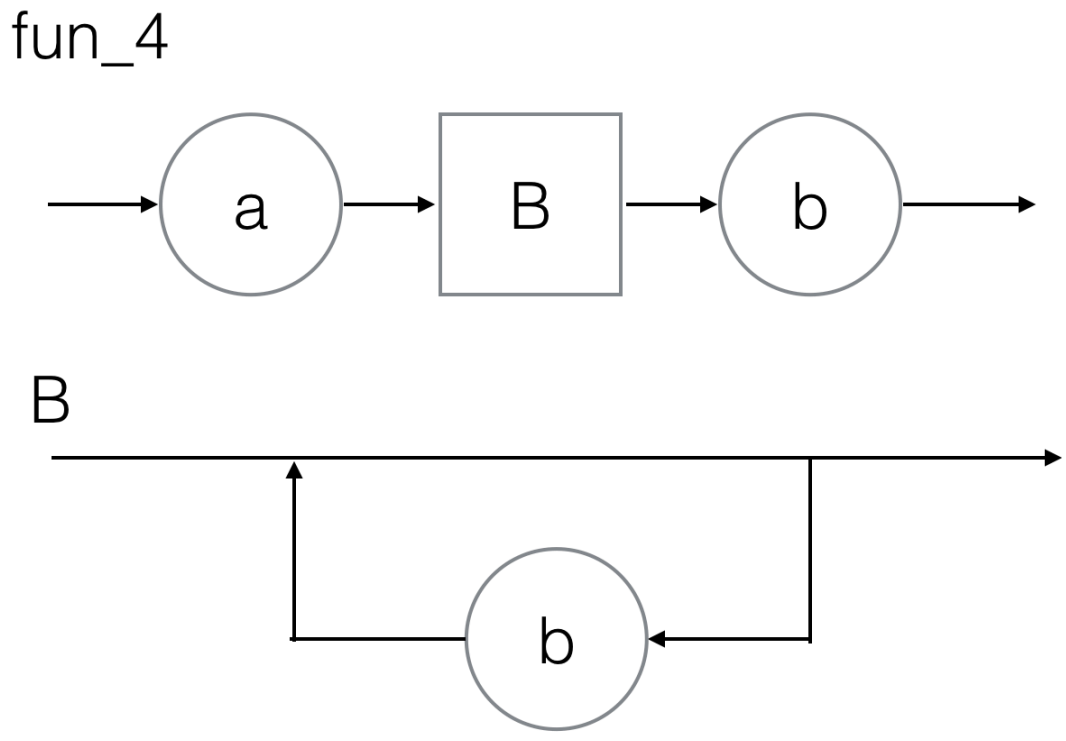
```
void B() {
    if(token == 'b')
        get_token();
    else
        error();
}

void C() {
    if(token == 'c')
        get_token();
    else
        error();
}
```

경우 3 의 의도는 parser 가 a 또는 b 또는 c 토큰을 인식하는 것이다. b 와 c 토큰을 인식할 때에는 각각 B 함수와 C 함수를 호출하여 토큰을 인식한다. 그러나 B 함수와 C 함수를 호출하는 방식으로는 프로그래밍을 구현하는 것이 불가능하다. parser 가 a 토큰을 인식하지 않으면 무조건 B()를 호출하므로 c 토큰은 인식하지 못하게 된다. 따라서 b 또는 c 중에 하나만 함수 호출 형태로 구현하는 것이 바람직하다.



```
void fun_3_1() {
    if(token == 'a')
        get_token();
    else if(token == 'b')
        get_token();
    else
        C();
}

void C() {
    if(token == 'c')
        get_token();
    else
        error();
}
```

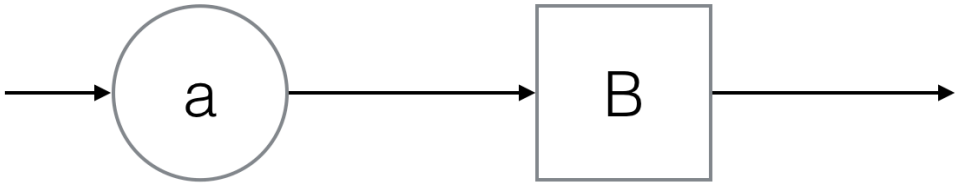
경우 4



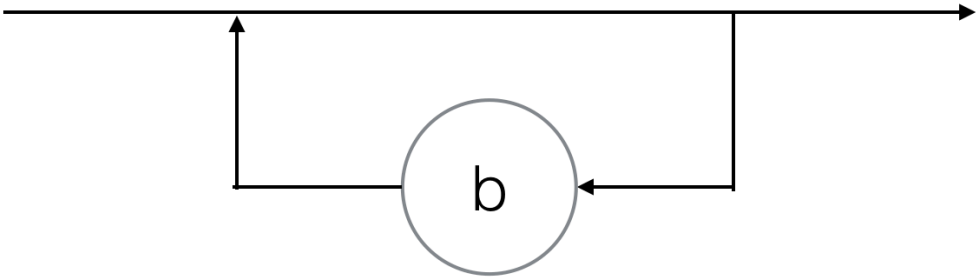
```
void fun_4() {
    if(token == 'a') {
        get_token();
        B();
        if(token == 'b')
            get_token();
        else error();
    }
    else error();
}

void B() {
    while(token == 'b')
        get_token();
}
```

경우 4 의 의도는 parser 가 처음에는 a 토큰을 인식한 후 B 함수를 호출하여 여러 개의 b 토큰을 인식한다. 그리고 마지막으로 한 개의 b 토큰을 인식한다. 그러나 이 syntax diagram 은 마지막에 존재하는 b 토큰에 의해 오류가 발생한다. B 함수 호출로 이미 모든 b 토큰을 인식했으므로 B 함수가 끝난 이후에 등장하는 b 토큰 조건에서는 인식할 수 있는 b 토큰이 존재하지 않는다. 따라서 마지막 b 토큰을 제거하는 것이 바람직하다.

fun_4_1



B



```
 void fun_4() {
    if(token == 'a') {
        get_token();
        B();
    }
    else error();
}

void B() {
    while(token == 'b')
        get_token();
}
```

(2) 0 과 1 로 이루어진 이진수를 입력으로 10 진수를 출력하는 lex 와 yacc 프로그램을 작성하시오. Lex 프로그램은 토큰으로 간단히 문자 한 개만을 잘라 보내도록 한다.

[yacc.y]

```
%{
#include <stdio.h>
void yyerror(char *s);
%}

%start number
%token ZERO ONE
%%
number: decimal {printf("%d₩n", $$);}
decimal: decimal binary {$$=$1*2 + $2;}
            | binary {$$=$1;}
binary: ZERO {$$=$1;}
            | ONE {$$=$1;}

%%
int main(void) {
    yyparse();
    return 0;
}

void yyerror(char *s) {
    printf("%s₩n", s);
}
```

[lex.l]

```
zero 0
one 1
empty [ ₩t₩n] /* 공백처리 */
%{
#include <stdio.h>
#include "y.tab.h"
extern int yylval;
int yylex();
int yywrap();
%}
%%
{zero} {yylval=0; return ZERO;}
{one} {yylval=1; return ONE;}
{empty} {return 0;}
%%
int yywrap() {
    return 1;
}
```

[실행결과]

```
msec@syssec:~/Desktop/SoongsilUniv/Exercise3$ ./binay2decimal
101011011101
2781
msec@syssec:~/Desktop/SoongsilUniv/Exercise3$ ./binay2decimal
100
4
msec@syssec:~/Desktop/SoongsilUniv/Exercise3$ ./binay2decimal
111
7
msec@syssec:~/Desktop/SoongsilUniv/Exercise3$ ./binay2decimal
1111011010101
7893
```

(3) C 언어 문법에서 declarator 및 parameter_type_list 는 어떻게 구성되어 있는지 설명하시오.

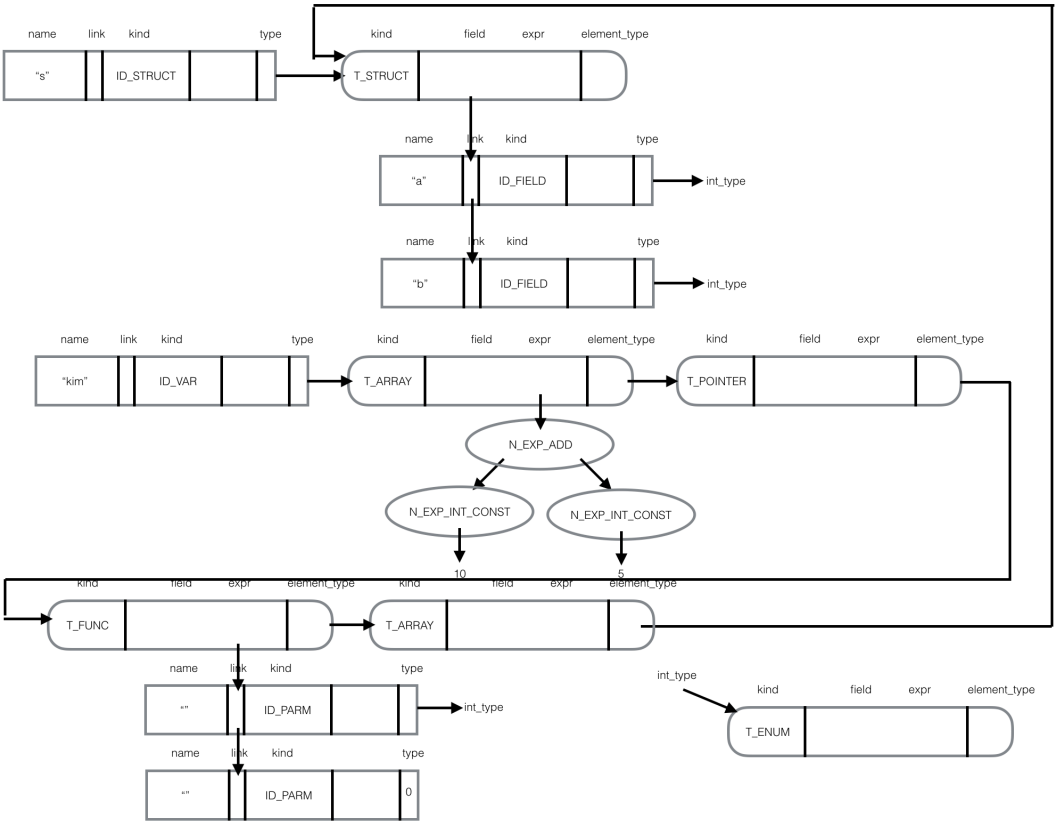
declarator
: pointer direct_declarator
| direct_declarator
pointer
: *
| * pointer
direct_declarator
: IDENTIFIER
| (declarator)
| direct_declarator [const_expression_opt]
| direct_declarator (parameter_type_list_opt)
const_expression_opt
: /* empty */
| const_expression
parameter_type_list_opt
: /* empty */
| parameter_type_list
parameter_type_list
: parameter_list
| parameter_list, …
parameter_list
: parameter_declaration
| parameter_list, parameter_declaration
parameter_declaration
: declaration_specifiers declarator
| declaration_specifiers abstract_declarator
| declaration_specifiers
abstract_declarator
: pointer
| direct_abstract_declarator
| pointer direct_abstract_declarator
direct_abstract_declarator
: (abstract_declarator)
| [const_expression_opt]
| (parameter_type_list_opt)
| direct_abstract_declarator [const_expression_opt]
| direct_abstract_declarator (parameter_type_list_opt)

declarator 는 주로 변수나 함수의 명칭, 혹은 타입 명칭을 선언하는데 사용한다. direct declarator 앞에 *를 여러 개 붙여 포인터 타입을 선언한다. direct declarator 는 한 개의 명칭으로 이루어져 있는 경우도 있고 direct declarator 뒤에 반복적으로 대괄호[]를 사용하는 배열형 타입과 direct declarator 뒤에 소괄호()를 사용하여 함수형 타입으로 선언한다.

parameter type list 는 여러 개의 parameter list 로 구성되는데, 이는 파라미터들과 각 파라미터의 타입을 선언하고 마지막에 …을 사용하여 그 개수를 유동적으로 선언할 수 있다.
각각의 parameter declaration 은 declaration specifiers 와 parameter list 로 구성되나 초기화는 허용을 하지 않는다. parameter declarator 는 declarator 와 같은 형식이지만, 그 명칭을 생략해서 쓸 수도 있는데 이를 abstract declarator 라고 한다. abstract declarator 는 명칭이 없이 단순히 파라미터의 개수나 배열 등의 타입만을 지정하여 사용하고자 할 때 사용된다.

(4) 다음의 선언문을 파싱하여 만들어지는 심볼테이블을 (타입테이블, 신택스 트리 및 이들의 연결 관계 및 테이블에 들어가는 내용을 포함하여) 그리시오.

struct s { int a, b; } (*kim[10 + 5]) (int, …)[];



(5) C 언어 프로그램에서 선언되는 함수명칭이 ID_FUNC 로 정해지는 시기와 방법, 이때 호출하는 함수와 하는 일을 간단히 설명하시오.

함수명칭이 ID_FUNC 로 정해지는 시기는 선언 명시자와 선언자 목록 분석을 마치고 일반선언문 분석을 마쳤을 때이다. 함수명칭을 ID_FUNC 로 정하기 위해 호출되는 함수는 setDeclaratorListSpecifier 함수이다. 심볼테이블 목록 <id>에 연결된 모든 심볼 테이블을 검사하고 명시자 테이블 <p>로부터 심볼 테이블의 속성을 결정하여 저장한다. <p>의 저장장소 명시자가 명칭 뒤에 소괄호()가 나와서 함수명칭인 것을 확인하면 함수명칭을 ID_FUNC 로 정한다.
- A_ID *setDeclaratorListSpecifier(A_ID *id, A_SPECIFIER *p);

함수명칭의 중복선언 여부 검사도 setDeclaratorListSpecifier 함수를 호출하여 진행한다. setDeclaratorListSpecifier 함수는 내부적으로 searchIdentifierAtCurrentLevel 함수를 호출하여 새로 입력된 함수명칭이 심볼테이블에 이미 저장된 명칭과 겹치는지 여부를 검사한다.
- A_ID *setDeclaratorListSpecifier(A_ID *id, A_SPECIFIER *p);
- A_ID *searchIdentifierAtCurrentLevel(char *s, A_ID *id);

함수명칭은 함수선언문을 분석할 때도 검사가 진행된다. 이때 호출되는 함수는 setFunctionDeclaratorSpecifier 함수이다. 주어진 심볼테이블 <id>의 명칭이 함수 타입으로 잘 선언되었는지 검사한다. 그리고 함수명칭이 앞서 선언되어 있는지 검사하여 함수 프로토타입으로 선언이 된 경우 그 프로토타입 선언 내용과 일치하는지와 중복선언이 되었는지 여부를 검사한다. 이때 중복선언여부 검사도 setFunctionDeclaratorSpecifier 함수 내부적으로 searchIdentifierAtCurrentLevel 을 호출하여 진행한다.
- A_ID *setFunctionDeclaratorSpecifier(A_ID *id, A_SPECIFIER *p);
- A_ID *searchIdentifierAtCurrentLevel(char *s, A_ID *id);

(6) 프로그램에서 선언되는 변수 명칭이 (그 종류가 ID_VAR 인 경우만) 심볼 테이블에 등록되고 중복선언되는지를 검사하는 시기와 방법을 설명하시오.

변수명칭이 심볼 테이블에 등록되는 시기는 declaration 에서 변수명칭을 갖는 declarator 부분이 나올 때이다. 변수명칭을 등록하기 위해서 makeIdentifier 함수를 호출한다. makeIdentifier 함수는 내부적으로 새로운 명칭이 들어오면 사용가능한 명칭 목록의 맨 앞에 추가한다.
- A_ID *makeIdentifier(char *s);

변수명칭의 중복선언 여부 검사는 setDeclaratorListSpecifier 함수를 호출하여 진행한다. setDeclaratorListSpecifier 함수는 내부적으로 searchIdentifierAtCurrentLevel 함수를 호출하여 새로 입력된 변수명칭이 심볼테이블에 이미 저장된 명칭과 겹치는지 여부를 검사한다.
- A_ID *setDeclaratorListSpecifier(A_ID *id, A_SPECIFIER *p);
- A_ID *searchIdentifierAtCurrentLevel(char *s, A_ID *id);

(7) 불완전참조란 무엇이고 이를 해결하는 시기와 방법을 간단히 설명하시오.

C 언어 프로그램 중 명령문 등에서 사용되는 명칭들은 미리 선언문 등을 통하여 선언되어 있어야 하며, 보통 그 선언문에서 속성을 완전하게 기술하여야 한다. 그러나 예외적으로 함수 명칭, 무조건 분기문(goto 문)의 레이블 명칭, 구조체나 나열형 태그 명칭 등의 경우에는 사용하기 전에 선언을 하지 않거나 혹은 명칭을 불완전하게 선언하고 사용할 수 있도록 허용하는데 이러한 참조를 불완전참조(Incomplete Reference) 또는 전방참조(Forward Reference)라고 한다.

함수 명칭을 미리 선언하지 않고 사용하는 경우에는 그 함수의 리턴 타입을 정수 타입으로 하여 몸체는 정의하지 않고 불완전하게 미리 선언한 것과 같이 처리한다. 함수 명칭은 사용 지점 이후에 혹은 외부 파일에서 함수 선언문으로 완전하게 정의되어야 한다.

goto 문에 나오는 레이블 명칭은 그 레이블의 위치가 분기 명령문 앞에 나오거나 혹은 나중에 해당함수 어디에라도 나오면 된다.

구조체 타입의 태크 명칭은 구조체의 필드 선언문 없이 불완전하게 선언될 수 있으나 반드시 원시프로그램 전방에 같은 스코프 범위 안에서 구조체 필드 선언문 목록을 갖추어 선언되어야 한다.

나열형 타입의 태크 명칭은 명칭상수 선언이 없이 불완전하게 선언될 수 있으나 반드시 원시프로그램 전방에 같은 스코프 범위 안에서 명칭상수 목록을 갖추어 선언되어야 한다.

불완전참조 문제는 전방참조되는 명칭이 나중에 적당한 스코프 범위에서 완전히 선언되었는지 여부를 검사하여 해결한다. 선언 여부 검사는 전체 프로그램의 끝부분, 복합문의 끝부분, 구조체의 필드 선언문 목록의 끝부분, 함수의 파라미터 선언문 목록의 끝부분에서 진행된다. 검사를 진행할 때 호출하는 함수는 checkForwardReference 함수이다.

- checkForwardReference()

(8) 다음의 명령문에 대한 신택스 트리를 그리시오.

```
switch (kim) {
   case 10 + 1: kim = kim * 2 + 1;
   default: for (i = 0; i < 5; i++);
}
```