# GCC LALR Syntax (2)
# C 언어를 위한 Yacc 문법

정원교

2004년 4월 24일

## 목 차

**제 1 절 22 주 문서를 시작하며** **2**

**제 2 절 Program 라벨** **2**

**제 3 절 Data type** **3**
3.1 Data type 이 받아들이는 문법 . . . . . 3
3.2 Data type 선언 . . . . . 25

**제 4 절 Expression** **33**
4.1 Operator 표현식 . . . . . 34
4.2 Statement 표현식 . . . . . 45
4.3 Declarator 표현식 . . . . . 57
4.4 Label 표현식 . . . . . 59

**제 5 절 Function** **61**

**제 6 절 22 주 문서를 마치며** **67**

# 제 1 절 22 주 문서를 시작하며

이번 주 문서에서는 GCC 에서 사용하는 C 용 LALR 문법에 대해서 다루게 됩니다. 하지만 이것을 어떻게 풀어서 설명을 할 수 있을까 많이 고민을 하였는데요, 결과적으로 각 data type 에 따라, 각 statement 에 따라, 마지막으로 함수에 대해 어떻게 LALR 문법이 진행되는지 살펴보도록 하겠습니다.

# 제 2 절 Program 라벨

GCC 에서 사용되는, $prefix/gcc/c-parse.in 파일에 선언되어 있는 yacc 문법의 시작은 program 라벨로 시작되는데, 이 라벨을 시작으로 외부 정의 라벨 (extdefs) 들로 수정이 되어진다. 아래부터는 (1) data type 과 (2) function type 으로 크게 나뉘어져 처리가 되어지며 각각에 대해서는 아래의 섹션에서 자세히 다루어질 것이다. 아래는 program 라벨에 대한 내용이다.

```
program: /* empty */
            { if (pedantic)
               pedwarn ("ISO C forbids an empty source file");
              finish_file ();
            }
      | extdefs
            {
              /* Closebrace 들을 빼놓았을 경우, global binding level
                 로 돌아간다.       */
              while (! global_bindings_p ())
                poplevel (0, 0, 0);
              finish_fname_decls ();
              finish_file ();
            }
      ;

/* 이 rule 에서 이상한 행동을 하는 이유는, datadef 를 통해
   도달한 notype_initdecls 는 type 의 유효 list 와 $0 내 sc
   specs 을 찾을 수 있기 때문이다.            */

extdefs:
      {$<ttype>$ = NULL_TREE; } extdef
      | extdefs {$<ttype>$ = NULL_TREE; ggc_collect(); } extdef
      ;

extdef:
      fndef
      | datadef
      | ASM_KEYWORD '(' expr ')' ';'
            { STRIP_NOPS ($3);
              if ((TREE_CODE ($3) == ADDR_EXPR
                   && TREE_CODE (TREE_OPERAND ($3, 0)) == STRING_CST)
                  || TREE_CODE ($3) == STRING_CST)
                assemble_asm ($3);
              else
                error ("argument of `asm' is not a constant string"); }
      | extension extdef
            { RESTORE_WARN_FLAGS ($1); }
      ;
```

# 제 3 절 Data type

Data type 이라고 하는 것은 의미에서 알 수 있듯이, C 언어에서 어떠한 정보를 담기 위한 그릇이라고 보면 될 것이다. GCC 의 LALR 문법에서는 크게 다음과 같이 나눌 수 있다. 첫째는 전역 변수이고, 둘째는 함수이다. 전역 변수를 해석하기 위한 것은 datadef 라는 부분이 전담한다. 함수의 정의는 fndef 가 전담하게 된다. 이 섹션에서는 data type 에 대해서 살펴볼 것이기 때문에 그에 대한 LALR 문법만 간략히 요약해보자. (말이 간략하게 요약한다는 것이지, 실제 보면 약간 어질어질할 것이다.)

## 3.1 Data type 이 받아들이는 문법

$prefix/gcc/c-parse.in 에 정의되어 있는 C 용 data type LALR 문법은 다음과 같이 크게 구분할 수 있다.

- 첫번째로는 Storage Class Specifier 를 들수 있겠다. 이것은 GCC 용으로 현재 function specifiers ("inline") 를 포함한다. 또한 static, extern, register, ... 등등에 대한 부분이 포함될 수 있다. 어떠한 변수의 type 에 대한 저장 형태를 지정해 주는데 사용된다.
- 두번째로는 Type Specifier 들 혹은 Type Qualifier 들로 들수 있다. 이에 대해서는 변수의 type 을 표현하는데 사용되는 int, float, double, short, long, ... 등등을 전체적으로 묶는데 사용된다.
- 세번째로는 Attribute 들을 들수 있다. GCC 에서 __attribute__ 혹은 __attribute 를 통해서 특정 변수에 대한 특성을 지정해줄 때 사용될 수 있기 때문에 이에 대한 처리 부분이 존재한다.
- 네번째로는 Declarator 를 말할 수 있는데, 즉 변수의 이름을 말한다. 변수의 이름은 단순히 abc, bcc, d_d, ... 등과 같은 것을 포함할 뿐만 아니라. abc[5], ab.c, ab→c, *abc, &arg , ... 등을 포함한다.
- 마지막 다섯번째로는 Initializer 인데, '=' 부터 시작되는 것을 처리하기 위해 존재하며, '=' 뒤에 올 수 있는 것은, 숫자, 문자열, statement 들이 올 수 있다.

아래에는 현재 data type 을 전체적으로 감싸고 있는 datadef 라벨에 대해 나열하였다.

```
datadef:
        setspecs notype_initdecls ';'
             { if (pedantic)
                error ("ISO C forbids data definition with no type or storage class");
              else if (!flag_traditional)
                warning ("data definition has no type or storage class");

              POP_DECLSPEC_STACK; }
      | declspecs_nots setspecs notype_initdecls ';'
             { POP_DECLSPEC_STACK; }
      | declspecs_ts setspecs initdecls ';'
             { POP_DECLSPEC_STACK; }
      | declspecs ';'
        { shadow_tag ($1); }
      | error ';'
      | error '}'
      | ';'
             { if (pedantic)
                pedwarn ("ISO C does not allow extra ';' outside of a function"); }
      ;
```

이에 대한 세부 사항들을 그룹별로 나눠서 설명을 계속 해야할 것같은데, datadef 라벨은 내부적으로 세 개의 그룹으로 나눌 수 가 있다. 하나는 declspecs* 로 시작하는 라벨, setspecs 라벨, *initdecls 라벨들이 그것이다. 위에서 언급한 (1) Storage Class Specifier, (2) Type Specifier 들 혹은 Type Qualifier, (3) Attribute 들을 인식하는데 사용되는 것이 declspecs* 로 시작하는 라벨들이다. 이러한 것들을 Declaration specifier 들의 목록이라고 부르는데, 아래와 같은 것이 존재한다.

- Storage class specifiers (SCSPEC), 이것은 GCC 용으로 현재 function specifiers ("inline") 를 포함한다.
- Type specifiers (typespec_*).
- Type qualifiers (TYPE_QUAL).
- Attribute specifier lists (attributes).

아래의 실제 문맥을 보면 알겠지만, 이것들은 TREE_LIST 로써 저장되는데, 목록의 head 는 specifier list 내 마지막 item 이다. 목록내의 각 entry 는 attribute specifier list 인 TREE_PURPOSE 나, single other specifier 혹은 qualifier 인 TREE_VALUE 나 목록의 나머지인 TREE_CHAIN 를 가지고 있다. TREE_STATIC 는 storage class specifier 혹은 attribute 가 다른 어떤 것이 많이 보일 경우 list 상에 설정된다; 이것은 목록의 시작에서 보다 다른 곳에서 storage class specifier 들의 쇠퇴하는 사용법에 대한 경고를 위해 사용된다. (이것을 제대로 하는 것은 storage class specifiers 로부터 분리적으로 다뤄지기 위해서 function specifiers 를 요구한다.

실제 내부 라벨들은 여러 종류의 case 별로 다음에 따라 분류되었다:

1. storage class specifier 가 포함되었느냐 안되었느냐; 문법의 몇몇 부분에서는 storage class specifiers (_sc 혹은 _nosc) 를 허락하지 않는다.
2. type specifier 가 보였는지 아닌지; type specifier 뒤, typedef name 는 (_ts 혹은 _nots) 를 재정의하기 위한 identifier 이다.
3. 목록이 attribute 로 시작하는지 아닌지; 특정 장소에서, 문법은 attribute (_sa 혹은 _nosa) 로 시작하지 않는 specifiers 를 요구한다.
4. 목록이 attribute (혹은 뒤로 따르는 어떤 attribute 가 해당 specifier 의 부분으로써 해석된 적이 있는 specifier) 로 끝나는지 아닌지; 이것은 attributes (_ea 혹은 _noea) 의 해석에서 shift-reduce conflict 를 피한다.

우선 Type specifier (Type qualifier 가 아닌) 에 대해서 먼저 살펴본다면, 이것은 typedef name 이 보이고 그것이 재선언되고 있을 경우, 이미 우리는 앞에서 이 선언을 본 적이 있다는 것이다. 아래의 라벨 이름 중 _reserved version 들은 예약된 단어로 시작하고 declaration specifiers 내 어느 위치에서든 나타날 수 있다; nonreserved version 들은 아마도 다른 어떤 type specifier 들 앞과 (이름들이 명명될 경우) 재선언되어진 뒤에서만 나타날 수 있다.

_attr 는 attribute 들로 끝나는 specifier 들을 의미하거나 어떠한 attribute 들이 specifier 의 부분으로써 해석될 수 있음을 의미한다. _nonattr 는 단순히 specifier 들이다. 그럼 typespec* 라벨에 대해서 보도록 하자.

```
typespec_nonattr:
        typespec_reserved_nonattr
      | typespec_nonreserved_nonattr
      ;

typespec_attr:
        typespec_reserved_attr
      ;

typespec_reserved_nonattr:
        TYPESPEC
            { OBJC_NEED_RAW_IDENTIFIER (1); }
      | structsp_nonattr
      ;

typespec_reserved_attr:
        structsp_attr
```

```
        ;

typespec_nonreserved_nonattr:
        TYPENAME
             { $$ = lookup_name ($1); }
      | TYPEOF '(' expr ')'
             { $$ = TREE_TYPE ($3); }
      | TYPEOF '(' typename ')'
             { $$ = groktypename ($3); }
      ;

/* structsp_attr: struct/union/enum specifier 들은 attribute 들로 끝날
   수도 혹은 뒤에 오는 어떤 attribute 들이 struct/union/enum specifier 의
   부분으로 해석될 수 도 있다.

  structsp_nonattr: other struct/union/enum specifier 들. */
structsp_attr:
        struct_head identifier '{'
             { $$ = start_struct (RECORD_TYPE, $2);
             }
        component_decl_list '}' maybe_attribute
             { $$ = finish_struct ($<ttype>4, $5, chainon ($1, $7)); }
      | struct_head '{' component_decl_list '}' maybe_attribute
             { $$ = finish_struct (start_struct (RECORD_TYPE, NULL_TREE),
                                   $3, chainon ($1, $5));
             }
      | union_head identifier '{'
             { $$ = start_struct (UNION_TYPE, $2); }
        component_decl_list '}' maybe_attribute
             { $$ = finish_struct ($<ttype>4, $5, chainon ($1, $7)); }
      | union_head '{' component_decl_list '}' maybe_attribute
             { $$ = finish_struct (start_struct (UNION_TYPE, NULL_TREE),
                                   $3, chainon ($1, $5));
             }
      | enum_head identifier '{'
             { $$ = start_enum ($2); }
        enumlist maybecomma_warn '}' maybe_attribute
             { $$ = finish_enum ($<ttype>4, nreverse ($5),
                                 chainon ($1, $8)); }
      | enum_head '{'
             { $$ = start_enum (NULL_TREE); }
        enumlist maybecomma_warn '}' maybe_attribute
             { $$ = finish_enum ($<ttype>3, nreverse ($4),
                                 chainon ($1, $7)); }
      ;

structsp_nonattr:
        struct_head identifier
             { $$ = xref_tag (RECORD_TYPE, $2); }
      | union_head identifier
             { $$ = xref_tag (UNION_TYPE, $2); }
      | enum_head identifier
             { $$ = xref_tag (ENUMERAL_TYPE, $2);
               if (pedantic && !COMPLETE_TYPE_P ($$))
                 pedwarn ("ISO C forbids forward references to 'enum' types"); }
      ;
```

```
struct_head:
        STRUCT
             { $$ = NULL_TREE; }
      | STRUCT attributes
             { $$ = $2; }
      ;

union_head:
        UNION
             { $$ = NULL_TREE; }
      | UNION attributes
             { $$ = $2; }
      ;

enum_head:
        ENUM
             { $$ = NULL_TREE; }
      | ENUM attributes
             { $$ = $2; }
      ;

component_decl_list:
        component_decl_list2
             { $$ = $1; }
      | component_decl_list2 component_decl
             { $$ = chainon ($1, $2);
               pedwarn ("no semicolon at end of struct or union"); }
      ;

component_decl_list2: /* empty */
             { $$ = NULL_TREE; }
      | component_decl_list2 component_decl ';'
             { $$ = chainon ($1, $2); }
      | component_decl_list2 ';'
             { if (pedantic)
                pedwarn ("extra semicolon in struct or union specified"); }
      ;

component_decl:
        declspecs_nosc_ts setspecs components
             { $$ = $3;
               POP_DECLSPEC_STACK; }
      | declspecs_nosc_ts setspecs save_filename save_lineno
             {
               if (pedantic)
                 pedwarn ("ISO C doesn't support unnamed structs/unions");

               $$ = grokfield($3, $4, NULL, current_declspecs, NULL_TREE);
               POP_DECLSPEC_STACK; }
      | declspecs_nosc_nots setspecs components_notype
             { $$ = $3;
               POP_DECLSPEC_STACK; }
      | declspecs_nosc_nots
             { if (pedantic)
                pedwarn ("ISO C forbids member declarations with no members");
               shadow_tag($1);
               $$ = NULL_TREE; }
```

```
        | error
              { $$ = NULL_TREE; }
        | extension component_decl
              { $$ = $2;
                RESTORE_WARN_FLAGS ($1); }
        ;

components:
          component_declarator
        | components ',' maybe_resetattrs component_declarator
              { $$ = chainon ($1, $4); }
        ;

components_notype:
          component_notype_declarator
        | components_notype ',' maybe_resetattrs component_notype_declarator
              { $$ = chainon ($1, $4); }
        ;

component_declarator:
          save_filename save_lineno declarator maybe_attribute
              { $$ = grokfield ($1, $2, $3, current_declspecs, NULL_TREE);
                decl_attributes (&$$, chainon ($4, all_prefix_attributes), 0); }
        | save_filename save_lineno
          declarator ':' expr_no_commas maybe_attribute
              { $$ = grokfield ($1, $2, $3, current_declspecs, $5);
                decl_attributes (&$$, chainon ($6, all_prefix_attributes), 0); }
        | save_filename save_lineno ':' expr_no_commas maybe_attribute
              { $$ = grokfield ($1, $2, NULL_TREE, current_declspecs, $4);
                decl_attributes (&$$, chainon ($5, all_prefix_attributes), 0); }
        ;

component_notype_declarator:
          save_filename save_lineno notype_declarator maybe_attribute
              { $$ = grokfield ($1, $2, $3, current_declspecs, NULL_TREE);
                decl_attributes (&$$, chainon ($4, all_prefix_attributes), 0); }
        | save_filename save_lineno
          notype_declarator ':' expr_no_commas maybe_attribute
              { $$ = grokfield ($1, $2, $3, current_declspecs, $5);
                decl_attributes (&$$, chainon ($6, all_prefix_attributes), 0); }
        | save_filename save_lineno ':' expr_no_commas maybe_attribute
              { $$ = grokfield ($1, $2, NULL_TREE, current_declspecs, $4);
                decl_attributes (&$$, chainon ($5, all_prefix_attributes), 0); }
        ;

enumlist:
          enumerator
        | enumlist ',' enumerator
              { if ($1 == error_mark_node)
                  $$ = $1;
                else
                  $$ = chainon ($3, $1); }
        | error
              { $$ = error_mark_node; }
        ;

extension:
```

```
        EXTENSION
190             { $$ = SAVE_WARN_FLAGS();
                  pedantic = 0;
                  warn_pointer_arith = 0;
                  warn_traditional = 0; }
        ;

195
identifier:
        IDENTIFIER
        | TYPENAME
        | OBJECTNAME
200     | CLASSNAME
        ;

maybecomma_warn:
          /* empty */
205     | ','
              { if (pedantic && ! flag_isoc99)
                  pedwarn ("comma at end of enumerator list"); }
        ;

210 /* 여기에서 강제적으로 readahead 를 하는 이유는 줄의 끝에 우리가 왔을 때, line
       과 file 은 yylex 가 다음 줄의 첫번째 token 을 읽수할 때까지 바뀌지 않기
       때문이다.       */
save_filename:
              { if (yychar == YYEMPTY)
215               yychar = YYLEX;
                $$ = input_filename; }
        ;

save_lineno:
220           { if (yychar == YYEMPTY)
                  yychar = YYLEX;
                $$ = lineno; }
        ;
```

위의 yacc 문법을 보면 아직은 알 수 없는 라벨들이 상당 수 존재한다는 것을 알 수 있을 것이며, 이로 인해 상당히 혼란스러울 수 있을 것이다. 좀 더 명확히 각각을 구분하기 위해서 나름대로 한 묶음으로 묶은 것이기 때문에 위에서도 그렇고, 앞으로 계속 yacc 문법이 나오겠는데, 그 묶은 내에서 RHS 상에서는 존재하지만 세부 기술되지 않은 문법들은 모두 나중에 볼 라벨들이며, 현재에서는 생각하지 않고 넘어가도 되는 것이기도 하다.

다음 declspecs* 라벨에 대해 살펴보겠는데, 이름이 모두 다음과 같은 순서로 나열되어 있다.

- SC (storage class)
- TS (type specifier)
- SA (start attribute)
- EA (end attribute)

각각의 요소가 존재하지는, 실제로 문맥상 나타났는지에 따라 앞에 no 를 붙일지 안붙일지를 결정하게 된다. 이 라벨들에 대한 것은 아래에 나열되어 있다.

```
declspecs_nosc_nots_nosa_noea:
        TYPE_QUAL
            { $$ = tree_cons (NULL_TREE, $1, NULL_TREE);
```

```
                TREE_STATIC ($$) = 1; }
      | declspecs_nosc_nots_nosa_noea TYPE_QUAL
            { $$ = tree_cons (NULL_TREE, $2, $1);
              TREE_STATIC ($$) = 1; }
      | declspecs_nosc_nots_nosa_ea TYPE_QUAL
            { $$ = tree_cons (NULL_TREE, $2, $1);
              TREE_STATIC ($$) = 1; }
      ;

declspecs_nosc_nots_nosa_ea:
        declspecs_nosc_nots_nosa_noea attributes
            { $$ = tree_cons ($2, NULL_TREE, $1);
              TREE_STATIC ($$) = TREE_STATIC ($1); }
      ;

declspecs_nosc_nots_sa_noea:
        declspecs_nosc_nots_sa_noea TYPE_QUAL
            { $$ = tree_cons (NULL_TREE, $2, $1);
              TREE_STATIC ($$) = 1; }
      | declspecs_nosc_nots_sa_ea TYPE_QUAL
            { $$ = tree_cons (NULL_TREE, $2, $1);
              TREE_STATIC ($$) = 1; }
      ;

declspecs_nosc_nots_sa_ea:
        attributes
            { $$ = tree_cons ($1, NULL_TREE, NULL_TREE);
              TREE_STATIC ($$) = 0; }
      | declspecs_nosc_nots_sa_noea attributes
            { $$ = tree_cons ($2, NULL_TREE, $1);
              TREE_STATIC ($$) = TREE_STATIC ($1); }
      ;

declspecs_nosc_ts_nosa_noea:
        typespec_nonattr
            { $$ = tree_cons (NULL_TREE, $1, NULL_TREE);
              TREE_STATIC ($$) = 1; }
      | declspecs_nosc_ts_nosa_noea TYPE_QUAL
            { $$ = tree_cons (NULL_TREE, $2, $1);
              TREE_STATIC ($$) = 1; }
      | declspecs_nosc_ts_nosa_ea TYPE_QUAL
            { $$ = tree_cons (NULL_TREE, $2, $1);
              TREE_STATIC ($$) = 1; }
      | declspecs_nosc_ts_nosa_noea typespec_reserved_nonattr
            { $$ = tree_cons (NULL_TREE, $2, $1);
              TREE_STATIC ($$) = 1; }
      | declspecs_nosc_ts_nosa_ea typespec_reserved_nonattr
            { $$ = tree_cons (NULL_TREE, $2, $1);
              TREE_STATIC ($$) = 1; }
      | declspecs_nosc_nots_nosa_noea typespec_nonattr
            { $$ = tree_cons (NULL_TREE, $2, $1);
              TREE_STATIC ($$) = 1; }
      | declspecs_nosc_nots_nosa_ea typespec_nonattr
            { $$ = tree_cons (NULL_TREE, $2, $1);
              TREE_STATIC ($$) = 1; }
      ;
```

```
declspecs_nosc_ts_nosa_ea:
        typespec_attr
             { $$ = tree_cons (NULL_TREE, $1, NULL_TREE);
               TREE_STATIC ($$) = 1; }
      | declspecs_nosc_ts_nosa_noea attributes
             { $$ = tree_cons ($2, NULL_TREE, $1);
               TREE_STATIC ($$) = TREE_STATIC ($1); }
      | declspecs_nosc_ts_nosa_noea typespec_reserved_attr
             { $$ = tree_cons (NULL_TREE, $2, $1);
               TREE_STATIC ($$) = 1; }
      | declspecs_nosc_ts_nosa_ea typespec_reserved_attr
             { $$ = tree_cons (NULL_TREE, $2, $1);
               TREE_STATIC ($$) = 1; }
      | declspecs_nosc_nots_nosa_noea typespec_attr
             { $$ = tree_cons (NULL_TREE, $2, $1);
               TREE_STATIC ($$) = 1; }
      | declspecs_nosc_nots_nosa_ea typespec_attr
             { $$ = tree_cons (NULL_TREE, $2, $1);
               TREE_STATIC ($$) = 1; }
      ;

declspecs_nosc_ts_sa_noea:
        declspecs_nosc_ts_sa_noea TYPE_QUAL
             { $$ = tree_cons (NULL_TREE, $2, $1);
               TREE_STATIC ($$) = 1; }
      | declspecs_nosc_ts_sa_ea TYPE_QUAL
             { $$ = tree_cons (NULL_TREE, $2, $1);
               TREE_STATIC ($$) = 1; }
      | declspecs_nosc_ts_sa_noea typespec_reserved_nonattr
             { $$ = tree_cons (NULL_TREE, $2, $1);
               TREE_STATIC ($$) = 1; }
      | declspecs_nosc_ts_sa_ea typespec_reserved_nonattr
             { $$ = tree_cons (NULL_TREE, $2, $1);
               TREE_STATIC ($$) = 1; }
      | declspecs_nosc_nots_sa_noea typespec_nonattr
             { $$ = tree_cons (NULL_TREE, $2, $1);
               TREE_STATIC ($$) = 1; }
      | declspecs_nosc_nots_sa_ea typespec_nonattr
             { $$ = tree_cons (NULL_TREE, $2, $1);
               TREE_STATIC ($$) = 1; }
      ;

declspecs_nosc_ts_sa_ea:
        declspecs_nosc_ts_sa_noea attributes
             { $$ = tree_cons ($2, NULL_TREE, $1);
               TREE_STATIC ($$) = TREE_STATIC ($1); }
      | declspecs_nosc_ts_sa_noea typespec_reserved_attr
             { $$ = tree_cons (NULL_TREE, $2, $1);
               TREE_STATIC ($$) = 1; }
      | declspecs_nosc_ts_sa_ea typespec_reserved_attr
             { $$ = tree_cons (NULL_TREE, $2, $1);
               TREE_STATIC ($$) = 1; }
      | declspecs_nosc_nots_sa_noea typespec_attr
             { $$ = tree_cons (NULL_TREE, $2, $1);
               TREE_STATIC ($$) = 1; }
      | declspecs_nosc_nots_sa_ea typespec_attr
             { $$ = tree_cons (NULL_TREE, $2, $1);
```

```
                TREE_STATIC ($$) = 1; }
        ;

declspecs_sc_nots_nosa_noea:
        SCSPEC
             { $$ = tree_cons (NULL_TREE, $1, NULL_TREE);
               TREE_STATIC ($$) = 0; }
      | declspecs_sc_nots_nosa_noea TYPE_QUAL
             { $$ = tree_cons (NULL_TREE, $2, $1);
               TREE_STATIC ($$) = 1; }
      | declspecs_sc_nots_nosa_ea TYPE_QUAL
             { $$ = tree_cons (NULL_TREE, $2, $1);
               TREE_STATIC ($$) = 1; }
      | declspecs_nosc_nots_nosa_noea SCSPEC
             { if (extra_warnings && TREE_STATIC ($1))
                warning ("`%s' is not at beginning of declaration",
                        IDENTIFIER_POINTER ($2));
               $$ = tree_cons (NULL_TREE, $2, $1);
               TREE_STATIC ($$) = TREE_STATIC ($1); }
      | declspecs_nosc_nots_nosa_ea SCSPEC
             { if (extra_warnings && TREE_STATIC ($1))
                warning ("`%s' is not at beginning of declaration",
                        IDENTIFIER_POINTER ($2));
               $$ = tree_cons (NULL_TREE, $2, $1);
               TREE_STATIC ($$) = TREE_STATIC ($1); }
      | declspecs_sc_nots_nosa_noea SCSPEC
             { if (extra_warnings && TREE_STATIC ($1))
                warning ("`%s' is not at beginning of declaration",
                        IDENTIFIER_POINTER ($2));
               $$ = tree_tscons (NULL_TREE, $2, $1);
               TREE_STATIC ($$) = TREE_STATIC ($1); }
      | declspecs_sc_nots_nosa_ea SCSPEC
             { if (extra_warnings && TREE_STATIC ($1))
                warning ("`%s' is not at beginning of declaration",
                        IDENTIFIER_POINTER ($2));
               $$ = tree_cons (NULL_TREE, $2, $1);
               TREE_STATIC ($$) = TREE_STATIC ($1); }
        ;

declspecs_sc_nots_nosa_ea:
        declspecs_sc_nots_nosa_noea attributes
             { $$ = tree_cons ($2, NULL_TREE, $1);
               TREE_STATIC ($$) = TREE_STATIC ($1); }
        ;

declspecs_sc_nots_sa_noea:
        declspecs_sc_nots_sa_noea TYPE_QUAL
             { $$ = tree_cons (NULL_TREE, $2, $1);
               TREE_STATIC ($$) = 1; }
      | declspecs_sc_nots_sa_ea TYPE_QUAL
             { $$ = tree_cons (NULL_TREE, $2, $1);
               TREE_STATIC ($$) = 1; }
      | declspecs_nosc_nots_sa_noea SCSPEC
             { if (extra_warnings && TREE_STATIC ($1))
                warning ("`%s' is not at beginning of declaration",
                        IDENTIFIER_POINTER ($2));
               $$ = tree_cons (NULL_TREE, $2, $1);
```

```
                TREE_STATIC ($$) = TREE_STATIC ($1); }
        | declspecs_nosc_nots_sa_ea SCSPEC
              { if (extra_warnings && TREE_STATIC ($1))
                  warning ("'%s' is not at beginning of declaration",
                           IDENTIFIER_POINTER ($2));
                $$ = tree_cons (NULL_TREE, $2, $1);
                TREE_STATIC ($$) = TREE_STATIC ($1); }
        | declspecs_sc_nots_sa_noea SCSPEC
              { if (extra_warnings && TREE_STATIC ($1))
                  warning ("'%s' is not at beginning of declaration",
                           IDENTIFIER_POINTER ($2));
                $$ = tree_cons (NULL_TREE, $2, $1);
                TREE_STATIC ($$) = TREE_STATIC ($1); }
        | declspecs_sc_nots_sa_ea SCSPEC
              { if (extra_warnings && TREE_STATIC ($1))
                  warning ("'%s' is not at beginning of declaration",
                           IDENTIFIER_POINTER ($2));
                $$ = tree_cons (NULL_TREE, $2, $1);
                TREE_STATIC ($$) = TREE_STATIC ($1); }
        ;

declspecs_sc_nots_sa_ea:
        declspecs_sc_nots_sa_noea attributes
              { $$ = tree_cons ($2, NULL_TREE, $1);
                TREE_STATIC ($$) = TREE_STATIC ($1); }
        ;

declspecs_sc_ts_nosa_noea:
        declspecs_sc_ts_notssa_noea TYPE_QUAL
              { $$ = tree_cons (NULL_TREE, $2, $1);
                TREE_STATIC ($$) = 1; }
        | declspecs_sc_ts_nosa_ea TYPE_QUAL
              { $$ = tree_cons (NULL_TREE, $2, $1);
                TREE_STATIC ($$) = 1; }
        | declspecs_sc_ts_nosa_noea typespec_reserved_nonattr
              { $$ = tree_cons (NULL_TREE, $2, $1);
                TREE_STATIC ($$) = 1; }
        | declspecs_sc_ts_nosa_ea typespec_reserved_nonattr
              { $$ = tree_cons (NULL_TREE, $2, $1);
                TREE_STATIC ($$) = 1; }
        | declspecs_sc_nots_nosa_noea typespec_nonattr
              { $$ = tree_cons (NULL_TREE, $2, $1);
                TREE_STATIC ($$) = 1; }
        | declspecs_sc_nots_nosa_ea typespec_nonattr
              { $$ = tree_cons (NULL_TREE, $2, $1);
                TREE_STATIC ($$) = 1; }
        | declspecs_nosc_ts_nosa_noea SCSPEC
              { if (extra_warnings && TREE_STATIC ($1))
                  warning ("'%s' is not at beginning of declaration",
                           IDENTIFIER_POINTER ($2));
                $$ = tree_cons (NULL_TREE, $2, $1);
                TREE_STATIC ($$) = TREE_STATIC ($1); }
        | declspecs_nosc_ts_nosa_ea SCSPEC
              { if (extra_warnings && TREE_STATIC ($1))
                  warning ("'%s' is not at beginning of declaration",
                           IDENTIFIER_POINTER ($2));
                $$ = tree_cons (NULL_TREE, $2, $1);
```

```
                TREE_STATIC ($$) = TREE_STATIC ($1); }
        | declspecs_sc_ts_nosa_noea SCSPEC
              { if (extra_warnings && TREE_STATIC ($1))
                  warning ("'%s' is not at beginning of declaration",
                           IDENTIFIER_POINTER ($2));
                $$ = tree_cons (NULL_TREE, $2, $1);
                TREE_STATIC ($$) = TREE_STATIC ($1); }
        | declspecs_sc_ts_nosa_ea SCSPEC
              { if (extra_warnings && TREE_STATIC ($1))
                  warning ("'%s' is not at beginning of declaration",
                           IDENTIFIER_POINTER ($2));
                $$ = tree_cons (NULL_TREE, $2, $1);
                TREE_STATIC ($$) = TREE_STATIC ($1); }
        ;

declspecs_sc_ts_nosa_ea:
        declspecs_sc_ts_nosa_noea attributes
              { $$ = tree_cons ($2, NULL_TREE, $1);
                TREE_STATIC ($$) = TREE_STATIC ($1); }
        | declspecs_sc_ts_nosa_noea typespec_reserved_attr
              { $$ = tree_cons (NULL_TREE, $2, $1);
                TREE_STATIC ($$) = 1; }
        | declspecs_sc_ts_nosa_ea typespec_reserved_attr
              { $$ = tree_cons (NULL_TREE, $2, $1);
                TREE_STATIC ($$) = 1; }
        | declspecs_sc_nots_nosa_noea typespec_attr
              { $$ = tree_cons (NULL_TREE, $2, $1);
                TREE_STATIC ($$) = 1; }
        | declspecs_sc_nots_nosa_ea typespec_attr
              { $$ = tree_cons (NULL_TREE, $2, $1);
                TREE_STATIC ($$) = 1; }
        ;

declspecs_sc_ts_sa_noea:
        declspecs_sc_ts_sa_noea TYPE_QUAL
              { $$ = tree_cons (NULL_TREE, $2, $1);
                TREE_STATIC ($$) = 1; }
        | declspecs_sc_ts_sa_ea TYPE_QUAL
              { $$ = tree_cons (NULL_TREE, $2, $1);
                TREE_STATIC ($$) = 1; }
        | declspecs_sc_ts_sa_noea typespec_reserved_nonattr
              { $$ = tree_cons (NULL_TREE, $2, $1);
                TREE_STATIC ($$) = 1; }
        | declspecs_sc_ts_sa_ea typespec_reserved_nonattr
              { $$ = tree_cons (NULL_TREE, $2, $1);
                TREE_STATIC ($$) = 1; }
        | declspecs_sc_nots_sa_noea typespec_nonattr
              { $$ = tree_cons (NULL_TREE, $2, $1);
                TREE_STATIC ($$) = 1; }
        | declspecs_sc_nots_sa_ea typespec_nonattr
              { $$ = tree_cons (NULL_TREE, $2, $1);
                TREE_STATIC ($$) = 1; }
        | declspecs_nosc_ts_sa_noea SCSPEC
              { if (extra_warnings && TREE_STATIC ($1))
                  warning ("'%s' is not at beginning of declaration",
                           IDENTIFIER_POINTER ($2));
                $$ = tree_cons (NULL_TREE, $2, $1);
```

```
                TREE_STATIC ($$) = TREE_STATIC ($1); }
        | declspecs_nosc_ts_sa_ea SCSPEC
              { if (extra_warnings && TREE_STATIC ($1))
                  warning ("'%s' is not at beginning of declaration",
                           IDENTIFIER_POINTER ($2));
                $$ = tree_cons (NULL_TREE, $2, $1);
                TREE_STATIC ($$) = TREE_STATIC ($1); }
        | declspecs_sc_ts_sa_noea SCSPEC
              { if (extra_warnings && TREE_STATIC ($1))
                  warning ("'%s' is not at beginning of declaration",
                           IDENTIFIER_POINTER ($2));
                $$ = tree_cons (NULL_TREE, $2, $1);
                TREE_STATIC ($$) = TREE_STATIC ($1); }
        | declspecs_sc_ts_sa_ea SCSPEC
              { if (extra_warnings && TREE_STATIC ($1))
                  warning ("'%s' is not at beginning of declaration",
                           IDENTIFIER_POINTER ($2));
                $$ = tree_cons (NULL_TREE, $2, $1);
                TREE_STATIC ($$) = TREE_STATIC ($1); }
        ;

declspecs_sc_ts_sa_ea:
          declspecs_sc_ts_sa_noea attributes
              { $$ = tree_cons ($2, NULL_TREE, $1);
                TREE_STATIC ($$) = TREE_STATIC ($1); }
        | declspecs_sc_ts_sa_noea typespec_reserved_attr
              { $$ = tree_cons (NULL_TREE, $2, $1);
                TREE_STATIC ($$) = 1; }
        | declspecs_sc_ts_sa_ea typespec_reserved_attr
              { $$ = tree_cons (NULL_TREE, $2, $1);
                TREE_STATIC ($$) = 1; }
        | declspecs_sc_nots_sa_noea typespec_attr
              { $$ = tree_cons (NULL_TREE, $2, $1);
                TREE_STATIC ($$) = 1; }
        | declspecs_sc_nots_sa_ea typespec_attr
              { $$ = tree_cons (NULL_TREE, $2, $1);
                TREE_STATIC ($$) = 1; }
        ;

declspecs_ts:
          declspecs_nosc_ts_nosa_noea
        | declspecs_nosc_ts_nosa_ea
        | declspecs_nosc_ts_sa_noea
        | declspecs_nosc_ts_sa_ea
        | declspecs_sc_ts_nosa_noea
        | declspecs_sc_ts_nosa_ea
        | declspecs_sc_ts_sa_noea
        | declspecs_sc_ts_sa_ea
        ;

declspecs_nots:
          declspecs_nosc_nots_nosa_noea
        | declspecs_nosc_nots_nosa_ea
        | declspecs_nosc_nots_sa_noea
        | declspecs_nosc_nots_sa_ea
        | declspecs_sc_nots_nosa_noea
        | declspecs_sc_nots_nosa_ea
```

```
        | declspecs_sc_nots_sa_noea
        | declspecs_sc_nots_sa_ea
        ;

declspecs_ts_nosa:
         declspecs_nosc_ts_nosa_noea
        | declspecs_nosc_ts_nosa_ea
        | declspecs_sc_ts_nosa_noea
        | declspecs_sc_ts_nosa_ea
        ;

declspecs_nots_nosa:
         declspecs_nosc_nots_nosa_noea
        | declspecs_nosc_nots_nosa_ea
        | declspecs_sc_nots_nosa_noea
        | declspecs_sc_nots_nosa_ea
        ;

declspecs_nosc_ts:
         declspecs_nosc_ts_nosa_noea
        | declspecs_nosc_ts_nosa_ea
        | declspecs_nosc_ts_sa_noea
        | declspecs_nosc_ts_sa_ea
        ;

declspecs_nosc_nots:
         declspecs_nosc_nots_nosa_noea
        | declspecs_nosc_nots_nosa_ea
        | declspecs_nosc_nots_sa_noea
        | declspecs_nosc_nots_sa_ea
        ;

declspecs_nosc:
         declspecs_nosc_ts_nosa_noea
        | declspecs_nosc_ts_nosa_ea
        | declspecs_nosc_ts_sa_noea
        | declspecs_nosc_ts_sa_ea
        | declspecs_nosc_nots_nosa_noea
        | declspecs_nosc_nots_nosa_ea
        | declspecs_nosc_nots_sa_noea
        | declspecs_nosc_nots_sa_ea
        ;

declspecs:
         declspecs_nosc_nots_nosa_noea
        | declspecs_nosc_nots_nosa_ea
        | declspecs_nosc_nots_sa_noea
        | declspecs_nosc_nots_sa_ea
        | declspecs_nosc_ts_nosa_noea
        | declspecs_nosc_ts_nosa_ea
        | declspecs_nosc_ts_sa_noea
        | declspecs_nosc_ts_sa_ea
        | declspecs_sc_nots_nosa_noea
        | declspecs_sc_nots_nosa_ea
        | declspecs_sc_nots_sa_noea
        | declspecs_sc_nots_sa_ea
        | declspecs_sc_ts_nosa_noea
```

```
        | declspecs_sc_ts_nosa_ea
        | declspecs_sc_ts_sa_noea
        | declspecs_sc_ts_sa_ea
        ;
```

그럼 이제, setspecs 라벨과 *initdecls 라벨에 대해서 알아 봐야 하는데, setspecs 라벨의 경우, 실제 LALR 문법상에서는 중요한 흐름을 담당하고 있지 않지만, LALR 문법을 떠나, 처리하는 과정에서의 역활을 어느정도 맡고 있다. 앞의 declspecs* 라벨들을 처리가 끝날 후, 즉 어떤 변수의 이름 (declarator) 을 선언하기 바로 전에 이 setspecs 라벨 처리가 이루어지게 되는데, 이는 뒤에 나올 declarator 들을 처리하는데 사용할 type 과 storage class spec 들을 기록하는데 사용된다. 전역변수 current_declspecs 의 outer-level value 들의 stack 을 유지하기 위해서 인데, function declarators 내에 내포된 (nested in) parm declaration 들의 이득을 위해서 이다. LALR 문법을 처리하는 과정에서 사용되는 전역 변수 및 각각에 대한 역활에 대한 것은 다른 섹션에서 언급될 것이다. 아래에 setspecs 라벨에 대해 나열되어 있다.

```
setspecs: /* empty */
            { pending_xref_error ();
              PUSH_DECLSPEC_STACK;
              split_specs_attrs ($<ttype>0,
                                 &current_declspecs, &prefix_attributes);
              all_prefix_attributes = prefix_attributes; }
        ;

setspecs_fp:
          setspecs
            { prefix_attributes = chainon (prefix_attributes, $<ttype>-2);
              all_prefix_attributes = prefix_attributes; }
        ;
```

이제 *initdecls 라벨로 넘어가면, 이제 이 라벨에서는 어떤 declarator 를 처리하기 위해서 존재한다. 아래의 각 라벨에서는 “초기화자(init) 라벨”에 대해서는 생략되어 있으며, 또한 “expr 라벨”에 대해서 생략되어 있는데, 이는 따로 란을 만들어 계속 설명할 것이다. 어떤 변수를 선언하다고 가정을 하였을 때, C 언어에서는 단순히 이름만 부여할 수도 있겠지만, 포인터 변수의 경우 앞에 ‘*’ 를 붙이는가 하면 때에 따라서는 앞에 추가적인 정보를 줄 수 있는데, 이러한 전체적인 부분을 담당하는 곳이 이 *initdecls 라벨 부분이라고 말할 수 있다.

```
notype_initdecls:
        notype_initdcl
        | notype_initdecls ',' maybe_resetattrs notype_initdcl
        ;

notype_initdcl:
          notype_declarator maybeasm maybe_attribute '='
            { $<ttype>$ = start_decl ($1, current_declspecs, 1,
                                      chainon ($3, all_prefix_attributes));
              start_init ($<ttype>$, $2, global_bindings_p ()); }
          init
            { finish_init ();
              finish_decl ($<ttype>5, $6, $2); }
        | notype_declarator maybeasm maybe_attribute
            { tree d = start_decl ($1, current_declspecs, 0,
                                   chainon ($3, all_prefix_attributes));
              finish_decl (d, NULL_TREE, $2); }
        ;

/* typespec_nonreserved_attr 는 존재하지 않는다.      */
```

```
initdecls:
       initdcl
       | initdecls ',' maybe_resetattrs initdcl
       ;

initdcl:
        declarator maybeasm maybe_attribute '='
             { $<ttype>$ = start_decl ($1, current_declspecs, 1,
                                       chainon ($3, all_prefix_attributes));
               start_init ($<ttype>$, $2, global_bindings_p ()); }
        init
             { finish_init ();
               finish_decl ($<ttype>5, $6, $2); }
       | declarator maybeasm maybe_attribute
             { tree d = start_decl ($1, current_declspecs, 0,
                                    chainon ($3, all_prefix_attributes));
               finish_decl (d, NULL_TREE, $2);
             }
       ;

/* 어떤 종류의 declarator (그래서, 모든 declarator 들은 explicit
   typespec 뒤에는 허락되었다.).         */
declarator:
        after_type_declarator
       | notype_declarator
       ;

/* explicit typespec 이든 아니든 허락하는 declarator.       이것들은
   typedef-name 를 재선언할 수 없다.        */
notype_declarator:
        notype_declarator '(' parmlist_or_identifiers %prec '.'
             { $$ = build_nt (CALL_EXPR, $1, $3, NULL_TREE); }
       | '(' maybe_attribute notype_declarator ')'
             { $$ = $2 ? tree_cons ($2, $3, NULL_TREE) : $3; }
       | '*' maybe_type_quals_attrs notype_declarator %prec UNARY
             { $$ = make_pointer_declarator ($2, $3); }
       | notype_declarator array_declarator %prec '.'
             { $$ = set_array_declarator_type ($2, $1, 0); }
       | IDENTIFIER
       ;

/* 오직 explicit typespec 뒤에서만 허락되는 declarator.        */
after_type_declarator:
        '(' maybe_attribute after_type_declarator ')'
             { $$ = $2 ? tree_cons ($2, $3, NULL_TREE) : $3; }
       | after_type_declarator '(' parmlist_or_identifiers %prec '.'
             { $$ = build_nt (CALL_EXPR, $1, $3, NULL_TREE); }
       | after_type_declarator array_declarator %prec '.'
             { $$ = set_array_declarator_type ($2, $1, 0); }
       | '*' maybe_type_quals_attrs after_type_declarator %prec UNARY
             { $$ = make_pointer_declarator ($2, $3); }
       | TYPENAME
       ;

/* 이것은 parmlist 혹은 identifier list 가 정상인 function definition
   내에서 사용된다.          이것의 값은 ..._TYPE node 의 목록 혹은 identifier 들의
   목록이다.      */
```

```
parmlist_or_identifiers:
        maybe_attribute
             { pushlevel (0);
               clear_parm_order ();
               declare_parm_level (1); }
        parmlist_or_identifiers_1
             { $$ = $3;
               parmlist_tags_warning ();
               poplevel (0, 0, 0); }
       ;

parmlist_or_identifiers_1:
        parmlist_1
      | identifiers ')'
             { tree t;
               for (t = $1; t; t = TREE_CHAIN (t))
                 if (TREE_VALUE (t) == NULL_TREE)
                   error ("`...' in old-style identifier list");
               $$ = tree_cons (NULL_TREE, NULL_TREE, $1);

               if ($<ttype>-1 != 0
                   && (TREE_CODE ($$) != TREE_LIST
                       || TREE_PURPOSE ($$) == 0
                       || TREE_CODE (TREE_PURPOSE ($$)) != PARM_DECL))
                 YYERROR1;
             }
       ;

/* Identifier 들의 nonempty list. */
identifiers:
       IDENTIFIER
             { $$ = build_tree_list (NULL_TREE, $1); }
      | identifiers ',' IDENTIFIER
             { $$ = chainon ($1, build_tree_list (NULL_TREE, $3)); }
       ;

/* 이것은 function declarator 내 parens 내에 보일 수 있는 것들이다.
   이것의 값은 ...TYPE node 들의 list 이다.        Attribute 들은
   ``void bar (int (__attribute__((__mode__(SI))) int foo));''
   와 같이 abstract declarator 내 open parenthesis 뒤에 나타나는 자신의 것과
   충돌을 피하기 위해서는 반드시 여기에 나타나야 한다.                          */
parmlist:
        maybe_attribute
             { pushlevel (0);
               clear_parm_order ();
               declare_parm_level (0); }
        parmlist_1
             { $$ = $3;
               parmlist_tags_warning ();
               poplevel (0, 0, 0); }
       ;

parmlist_1:
        parmlist_2 ')'
      | parms ';'
             { tree parm;
               if (pedantic)
```

```
                pedwarn ("ISO C forbids forward parameter declarations");
              for (parm = getdecls (); parm; parm = TREE_CHAIN (parm))
                TREE_ASM_WRITTEN (parm) = 1;
              clear_parm_order (); }
        maybe_attribute
              { }
        parmlist_1
              { $$ = $6; }
      | error ')'
              { $$ = tree_cons (NULL_TREE, NULL_TREE, NULL_TREE); }
      ;

/* 이것은 function declarator 내 parens 내에 보일 수 있는 것들이다.
   이것의 값은 grokdeclarator 가 예상하고 있는 format 에 맞춰진다.                    */
parmlist_2: /* empty */
              { $$ = get_parm_info (0); }
      | ELLIPSIS
              { $$ = get_parm_info (0);
                error ("ISO C requires a named argument before '...'");
              }
      | parms
              { $$ = get_parm_info (1); }
      | parms ',' ELLIPSIS
              { $$ = get_parm_info (0); }
      ;

parms:
      firstparm
              { push_parm_decl ($1); }
      | parms ',' parm
              { push_parm_decl ($3); }
      ;

/* parmlist 내에서 발견된 single parameter declaration 혹은
   parameter type name. */
parm:
       declspecs_ts setspecs parm_declarator maybe_attribute
              { $$ = build_tree_list (build_tree_list (current_declspecs,
                                                       $3),
                                      chainon ($4, all_prefix_attributes));
                POP_DECLSPEC_STACK; }
      | declspecs_ts setspecs notype_declarator maybe_attribute
              { $$ = build_tree_list (build_tree_list (current_declspecs,
                                                       $3),
                                      chainon ($4, all_prefix_attributes));
                POP_DECLSPEC_STACK; }
      | declspecs_ts setspecs absdcl_maybe_attribute
              { $$ = $3;
                POP_DECLSPEC_STACK; }
      | declspecs_nots setspecs notype_declarator maybe_attribute
              { $$ = build_tree_list (build_tree_list (current_declspecs,
                                                       $3),
                                      chainon ($4, all_prefix_attributes));
                POP_DECLSPEC_STACK; }

      | declspecs_nots setspecs absdcl_maybe_attribute
              { $$ = $3;
```

```
                POP_DECLSPEC_STACK; }
        ;

/* 첫번째 parm 이며, 이것은 반드시 parser stack 의 꼭대기로부터 attribute 들을
   빨아내야 한다.        */
firstparm:
        declspecs_ts_nosa setspecs_fp parm_declarator maybe_attribute
                { $$ = build_tree_list (build_tree_list (current_declspecs,
                                                         $3),
                                        chainon ($4, all_prefix_attributes));
                POP_DECLSPEC_STACK; }
        | declspecs_ts_nosa setspecs_fp notype_declarator maybe_attribute
                { $$ = build_tree_list (build_tree_list (current_declspecs,
                                                         $3),
                                        chainon ($4, all_prefix_attributes));
                POP_DECLSPEC_STACK; }
        | declspecs_ts_nosa setspecs_fp absdcl_maybe_attribute
                { $$ = $3;
                POP_DECLSPEC_STACK; }
        | declspecs_nots_nosa setspecs_fp notype_declarator maybe_attribute
                { $$ = build_tree_list (build_tree_list (current_declspecs,
                                                         $3),
                                        chainon ($4, all_prefix_attributes));
                POP_DECLSPEC_STACK; }

        | declspecs_nots_nosa setspecs_fp absdcl_maybe_attribute
                { $$ = $3;
                POP_DECLSPEC_STACK; }
        ;

/* notype_declarator 에 추가적으로 parameter list 내에 나타날 수 있는
   declarator 의 종류들.    이것은 after_type_declarator 와 비슷하지만 identifier
   로서별s 괄호 내 typedef name 이 오는 것을 허락하지 않는다.                (왜냐하면 arg
   로 typedef 를 가지는 function 과 충돌할 수 있기 때문이다.)                 */
parm_declarator:
        parm_declarator_starttypename
        | parm_declarator_nostarttypename
        ;

parm_declarator_starttypename:
        parm_declarator_starttypename '(' parmlist_or_identifiers %prec '.'
                { $$ = build_nt (CALL_EXPR, $1, $3, NULL_TREE); }
        | parm_declarator_starttypename array_declarator %prec '.'
                { $$ = set_array_declarator_type ($2, $1, 0); }
        | TYPENAME
        ;

parm_declarator_nostarttypename:
        parm_declarator_nostarttypename '(' parmlist_or_identifiers %prec '.'
                { $$ = build_nt (CALL_EXPR, $1, $3, NULL_TREE); }
        | parm_declarator_nostarttypename array_declarator %prec '.'
                { $$ = set_array_declarator_type ($2, $1, 0); }
        | '*' maybe_type_quals_attrs parm_declarator_starttypename %prec UNARY
                { $$ = make_pointer_declarator ($2, $3); }
        | '*' maybe_type_quals_attrs parm_declarator_nostarttypename %prec UNARY
                { $$ = make_pointer_declarator ($2, $3); }
        | '(' maybe_attribute parm_declarator_nostarttypename ')'
```

```
                { $$ = $2 ? tree_cons ($2, $3, NULL_TREE) : $3; }
        ;

absdcl: /* an absolute declarator */
        /* empty */
                { $$ = NULL_TREE; }
        | absdcl1
        ;

absdcl_maybe_attribute:
        /* empty */
                { $$ = build_tree_list (build_tree_list (current_declspecs,
                                                         NULL_TREE),
                                        all_prefix_attributes); }
        | absdcl1
                { $$ = build_tree_list (build_tree_list (current_declspecs,
                                                         $1),
                                        all_prefix_attributes); }
        | absdcl1_noea attributes
                { $$ = build_tree_list (build_tree_list (current_declspecs,
                                                         $1),
                                        chainon ($2, all_prefix_attributes)); }
        ;

absdcl1: /* a nonempty absolute declarator */
         absdcl1_ea
        | absdcl1_noea
        ;

absdcl1_noea:
         direct_absdcl1
        | '*' maybe_type_quals_attrs absdcl1_noea
                { $$ = make_pointer_declarator ($2, $3); }
        ;

absdcl1_ea:
         '*' maybe_type_quals_attrs
                { $$ = make_pointer_declarator ($2, NULL_TREE); }
        | '*' maybe_type_quals_attrs absdcl1_ea
                { $$ = make_pointer_declarator ($2, $3); }
        ;

direct_absdcl1:
         '(' maybe_attribute absdcl1 ')'
                { $$ = $2 ? tree_cons ($2, $3, NULL_TREE) : $3; }
        | direct_absdcl1 '(' parmlist
                { $$ = build_nt (CALL_EXPR, $1, $3, NULL_TREE); }
        | direct_absdcl1 array_declarator
                { $$ = set_array_declarator_type ($2, $1, 1); }
        | '(' parmlist
                { $$ = build_nt (CALL_EXPR, NULL_TREE, $2, NULL_TREE); }
        | array_declarator
                { $$ = set_array_declarator_type ($1, NULL_TREE, 1); }
        ;

/* Array type 을 위한 declarator 의 [...] 부분.  */
array_declarator:
```

```
        '[' expr ']'
             { $$ = build_array_declarator ($2, NULL_TREE, 0, 0); }
      | '[' declspecs_nosc expr ']'
             { $$ = build_array_declarator ($3, $2, 0, 0); }
      | '[' ']'
             { $$ = build_array_declarator (NULL_TREE, NULL_TREE, 0, 0); }
      | '[' declspecs_nosc ']'
             { $$ = build_array_declarator (NULL_TREE, $2, 0, 0); }
      | '[' '*' ']'
             { $$ = build_array_declarator (NULL_TREE, NULL_TREE, 0, 1); }
      | '[' declspecs_nosc '*' ']'
             { $$ = build_array_declarator (NULL_TREE, $2, 0, 1); }
      | '[' SCSPEC expr ']'
             { if (C_RID_CODE ($2) != RID_STATIC)
                error ("storage class specifier in array declarator");
               $$ = build_array_declarator ($3, NULL_TREE, 1, 0); }
      | '[' SCSPEC declspecs_nosc expr ']'
             { if (C_RID_CODE ($2) != RID_STATIC)
                error ("storage class specifier in array declarator");
               $$ = build_array_declarator ($4, $3, 1, 0); }
      | '[' declspecs_nosc SCSPEC expr ']'
             { if (C_RID_CODE ($3) != RID_STATIC)
                error ("storage class specifier in array declarator");
               $$ = build_array_declarator ($4, $2, 1, 0); }
      ;

maybe_resetattrs:
        maybe_attribute
             { all_prefix_attributes = chainon ($1, prefix_attributes); }
      ;

maybecomma:
        /* empty */
      | ','
      ;

maybeasm:
        /* empty */
             { $$ = NULL_TREE; }
      | ASM_KEYWORD '(' string ')'
             { if (TREE_CHAIN ($3)) $3 = combine_strings ($3);
               $$ = $3;
             }
      ;

maybe_attribute:
    /* empty */
             { $$ = NULL_TREE; }
      | attributes
             { $$ = $1; }
      ;

attributes:
    attribute
             { $$ = $1; }
      | attributes attribute
             { $$ = chainon ($1, $2); }
```

```
      ;

attribute:
     ATTRIBUTE '(' '(' attribute_list ')' ')'
            { $$ = $4; }
      ;

attribute_list:
     attrib
            { $$ = $1; }
       | attribute_list ',' attrib
            { $$ = chainon ($1, $3); }
      ;

attrib:
   /* empty */
            { $$ = NULL_TREE; }
       | any_word
            { $$ = build_tree_list ($1, NULL_TREE); }
       | any_word '(' IDENTIFIER ')'
            { $$ = build_tree_list ($1, build_tree_list (NULL_TREE, $3)); }
       | any_word '(' IDENTIFIER ',' nonnull_exprlist ')'
            { $$ = build_tree_list ($1, tree_cons (NULL_TREE, $3, $5)); }
       | any_word '(' exprlist ')'
            { $$ = build_tree_list ($1, $3); }
      ;

any_word:
        identifier
       | SCSPEC
       | TYPESPEC
       | TYPE_QUAL
      ;
```

이제 "초기화자"에 대해서 살펴보도록 하자. 초기화자란 initializer 를 한글 그대로 번역한 것인데, 어떤 변수에 '=' 을 이용하여 초기화하고자 할 때 넣는 값들을 말한다. 이 문법에서는 값들을 읽어 들이는 과정에 대한 yacc 문법을 살펴보도록 하겠다.

```
init:
      expr_no_commas
       | '{'
            { really_start_incremental_init (NULL_TREE); }
        initlist_maybe_comma '}'
            { $$ = pop_init_level (0); }
       | error
            { $$ = error_mark_node; }
      ;

/* 'initlist_maybe_comma' 는 중괄호들내 초기화자의 알맹이들이다.                */
initlist_maybe_comma:
        /* empty */
            { if (pedantic)
               pedwarn ("ISO C forbids empty initializer braces"); }
       | initlist1 maybecomma
      ;
```

```
initlist1:
        initelt
      | initlist1 ',' initelt
      ;

/* 'initelt' 는 초기화자의 single element 이다.
   중괄호를 사용할 것이다.              */
initelt:
        designator_list '=' initval
             { if (pedantic && ! flag_isoc99)
                pedwarn ("ISO C89 forbids specifying subobject to initialize"); }
      | designator initval
             { if (pedantic)
                pedwarn ("obsolete use of designated initializer without '='"); }
      | identifier ':'
             { set_init_label ($1);
               if (pedantic)
                 pedwarn ("obsolete use of designated initializer with ':'"); }
        initval
      | initval
      ;

initval:
        '{'
             { push_init_level (0); }
        initlist_maybe_comma '}'
             { process_init_element (pop_init_level (0)); }
      | expr_no_commas
             { process_init_element ($1); }
      | error
      ;

designator_list:
        designator
      | designator_list designator
      ;

designator:
        '.' identifier
             { set_init_label ($2); }
      /* Labeled element 들을 위해 존재한다.        array element 초기화자를 위한
         문법은 Objective-C message 를 위한 문법과 충돌하여서 Objective-C
         grammar 에서 아래의 것을 포함하지 않도록 하였다.                    */
      | '[' expr_no_commas ELLIPSIS expr_no_commas ']'
             { set_init_index ($2, $4);
               if (pedantic)
                 pedwarn ("ISO C forbids specifying range of elements to initialize"); }
      | '[' expr_no_commas ']'
             { set_init_index ($2, NULL_TREE); }
      ;
```

위의 "초기화자" 관련 라벨에 대해서 살펴보면 알겠지만, expr_no_commas 라벨에 대해서는 빠져있음을 확인할 수 있다. 이 라벨의 경우 expr 관련 라벨이기 때문에 다음 섹션에서 자세히 언급될 수 있을 것이다. 물론 data type 을 선언할 때 expr 을 내부적으로 사용할 수 있다는 말도 된다.

## 3.2 Data type 선언

이제부터 각 data type 을 선언한다고 했을 때, 실제로 GCC 에서는 유형별로 어떻게 처리가 되는지 알아보도록 하자. Yacc 문법이 GCC 에서 어떻게 처리되는지에 대한 정보는 yydebug () 함수를 통해서 볼 수 있으며, −dy 옵션을 이용하여 실제 처리 과정을 확인할 수 있다. 아래는 단순한 int i; 라고 선언했을 때 어떻게 처리되는지에 대한 −dy 옵션 활성화시 화면이다.

```
aso@weongyo:~/gcc-3.1/gcc$ ./cc1 -dy
int i;
Starting parse
Entering state 0
Reading a token: Next token is 260 (TYPESPEC [CPP_NAME] ‘int’)
Reducing via rule 3 (line 306),  -> @1
state stack now 0
Entering state 2
Next token is 260 (TYPESPEC [CPP_NAME] ‘int’)
Shifting token 260 (TYPESPEC), Entering state 7
Reducing via rule 263 (line 1271), TYPESPEC  -> typespec_reserved_nonattr
state stack now 0 2
Entering state 42
Reducing via rule 260 (line 1262), typespec_reserved_nonattr  -> typespec_nonattr
state stack now 0 2
Entering state 40
Reducing via rule 133 (line 861), typespec_nonattr  -> declspecs_nosc_ts_nosa_noea
state stack now 0 2
Entering state 25
Reading a token: Next token is 257 (IDENTIFIER [CPP_NAME] ‘i’)
Reducing via rule 202 (line 1153), declspecs_nosc_ts_nosa_noea  -> declspecs_ts
state stack now 0 2
Entering state 37
Reducing via rule 117 (line 743),  -> setspecs
state stack now 0 2 37
Entering state 140
Next token is 257 (IDENTIFIER [CPP_NAME] ‘i’)
Shifting token 257 (IDENTIFIER), Entering state 62
Reducing via rule 346 (line 1601), IDENTIFIER  -> notype_declarator
state stack now 0 2 37 140
Entering state 222
Reading a token: Next token is 59 (’;’ [;])
Reducing via rule 326 (line 1530), notype_declarator  -> declarator
state stack now 0 2 37 140
Entering state 220
Next token is 59 (’;’ [;])
Reducing via rule 273 (line 1304),  -> maybeasm
state stack now 0 2 37 140 220
Entering state 314
Next token is 59 (’;’ [;])
Reducing via rule 281 (line 1345),  -> maybe_attribute
state stack now 0 2 37 140 220 314
Entering state 417
Next token is 59 (’;’ [;])
Reducing via rule 277 (line 1322), declarator maybeasm maybe_attribute  -> initdcl
state stack now 0 2 37 140
Entering state 219
Reducing via rule 269 (line 1294), initdcl  -> initdecls
state stack now 0 2 37 140
Entering state 218
```

```
Next token is 59 (';' [;])
Shifting token 59 (';'), Entering state 310
Reducing via rule 13 (line 336), declspecs_ts setspecs initdecls ';'  -> datadef
state stack now 0 2
Entering state 18
Reducing via rule 8 (line 313), datadef  -> extdef
state stack now 0 2
Entering state 17
Reducing via rule 4 (line 308), @1 extdef  -> extdefs
state stack now 0
Entering state 1
Reading a token: Now at end of input.
Reducing via rule 2 (line 291), extdefs  -> program
state stack now 0
Entering state 898
Now at end of input.
Shifting token 0 ($), Entering state 899
Now at end of input.

Execution times (seconds)
 lexical analysis      :   0.00 ( 0%) usr   0.00 ( 0%) sys   0.01 ( 1%) wall
 parser                :   0.00 ( 0%) usr   0.01 (33%) sys   0.04 ( 3%) wall
 varconst              :   0.00 ( 0%) usr   0.00 ( 0%) sys   0.01 ( 1%) wall
 symout                :   0.00 ( 0%) usr   0.01 (33%) sys   0.01 ( 1%) wall
 TOTAL                 :   0.02             0.03             1.33
aso@weongyo:~/gcc-3.1/gcc$
```

아래부터는 C 언어에서 주로 사용되는 대표적인 변수 선언문이 GCC 에서 어떠한 방향으로 Yacc 문법이 처리되어지고 있는지 각각 알아보도록 하겠다. 그리고 이러한 주요 변수를 처리하면서 GCC 내부에서 어떠한 operation 들이 이루어지는지에 대해서도 언급하도록 하자.

**(1)** int i;

첫번째 예제로써 많이 선언되는 int i; 에 대해서 간단히 살펴보도록 하자. 아래는 하향적인 모습으로 yacc 문법의 진행에 대해서 언급하였는데, 실제로는 상향식 해석 방식을 취하고 있다는 것은 여러분들도 알고 있을 것이다.

```
program -> extdefs
  extdefs -> extdef
    extdef -> datadef
      datadef -> declspecs_ts setspecs initdecls ';'
        declspecs_ts -> declspecs_nosc_ts_nosa_noea
          declspecs_nosc_ts_nosa_noea -> typespec_nonattr
            typespec_nonattr -> typespec_reserved_nonattr
              typespec_reserved_nonattr -> TYPESPEC
        setspecs
        initdecls -> initdcl
          initdcl -> declarator maybeasm maybe_attribute
            declarator -> notype_declarator
              notype_declarator -> IDENTIFIER
            maybeasm
            maybe_attribute
```

'int' 는 TYPESPEC 으로 해석이 되고, 변수 이름 'i' 는 IDENTIFIER 로써 해석된다는 사실을 할 수 있을 것이다. declspecs_* 라벨에서 이루어지는 것은 하위 라벨들의 yyval 를 tree TREE_LIST node 의 각각의 값 purpose, value, chain 에 넣는 것과, 경우에 따라 TREE_STATIC ($$) 의 값을 1 로 설정하는 것이다.

여기까지 하였으면, setspecs 라벨의 내부 수행이 이루어지게 되는데, 여기서의 목적은 $prefix/gcc/c-parse.in 파일내에 선언되어 있는 전역변수 prefix_attributes 와 all_prefix_attributes, current_declspecs 를 declspec_stack 에 TREE_LIST 형식으로 stack 을 만들어 PUSH 하는 것이다. PUSH 를 위해서 PUSH_DECLSPEC_STACK 매크로를 이용한다. GCC 의 yacc 문법에서 사용되는 전역 변수에 대해서는 후에 설명을 하도록 하겠다.

이제 변수 이름 'i' 를 해석할 차례이며, initdcl 라벨의 내부 수행이 이루어진다. 'int i' 까지 선언이 되었을 경우 실제로 이것이 변수 선언임을 인식할 수 있게 되며 start_decl () 함수를 호출함으로써 실제 declarator 를 만들게 되며, 선언이 완료되었을 경우 finish_decl () 함수를 호출하게 된다.

이제 하나의 완전한 declarator 를 처리 하였기 때문에 POP_DECLSPEC_STACK 매크로를 이용하여 PUSH 된 것들을 POP 한다.

이제 우리가 입력한 모든 줄을 해석하였기 때문에, 최상위 라벨인 program 라벨에 선언되어 있는 내부 operation 들이 수행하며, 수행되는 함수로써는 finish_fname_decls () 와 finish_file () 가 있다.

실제 각 함수의 내부 수행에 대해서는 다음 주 강의에서 선언의 유형별로 더 자세히 살펴보도록 하겠다.

**(2)** int a, b, c=1;

두번째로는, 하나의 type 에 여러개의 변수를 선언하고, 끝에서는 변수의 값을 초기화하는 모형을 예제로 나타내었다. 다시 한번 더 언급하지만, 아래의 그래프는 하향식 방법으로 나타내었지, 실제 내부 operation 의 실행 순서하고는 관계가 없다.

```
program -> extdefs
  extdefs -> extdef
    extdef -> datadef
      datadef -> declspecs_ts setspecs initdecls ';'
        declspecs_ts -> declspecs_nosc_ts_nosa_noea
          declspecs_nosc_ts_nosa_noea -> typespec_nonattr
            typespec_nonattr -> typespec_reserved_nonattr
              typespec_reserved_nonattr -> TYPESPEC
        setspecs
        initdecls -> initdecls ',' maybe_resetattrs initdcl
          initdecls -> initdecls ',' maybe_resetattrs initdcl
            initdecls -> initdcl
              initdcl -> declarator maybeasm maybe_attribute
                declarator -> notype_declarator
                notype_declarator -> IDENTIFIER
                maybeasm
                maybe_attribute
            maybe_resetattrs -> maybe_attribute
              maybe_attribute
            initdcl -> declarator maybeasm maybe_attribute
              declarator -> notype_declarator
                notype_declarator -> IDENTIFIER
              maybeasm
              maybe_attribute
          maybe_resetattrs -> maybe_attribute
            maybe_attribute
          initdcl -> declarator maybeasm maybe_attribute '=' init
            declarator -> notype_declarator
              notype_declarator -> IDENTIFIER
            maybeasm
            maybe_attribute
```

```
init -> expr_no_commas
  expr_no_commas -> cast_expr
    cast_expr -> unary_expr
      unary_expr -> primary
        primary -> CONSTANT
```

declspecs_ts 라벨과 setspecs 이 실행되는 과정은 위의 예제 (1) 과 같은 방법으로 실행이 된다. 눈여겨봐야 할 부분은 initdecls 라벨 하위부터 인데, 우선 변수가 한 개 이상 선언되어 있다는 점과 끝 변수는 초기화자가 존재한다는 사실이다. 이제 하나씩 살펴보면, 변수 'a' 가 선언될 때 뒤의 token 이 ',' 이기 때문에 start_decl () 함수와 finish_decl () 함수가 실행된다는 사실을 알 수 있으며, 이는 변수 'b' 가 선언될 때도 마찬가지이다. 이미 type 에 대한 정보는 전역변수 current_declspecs 와 all_prefix_attributes 들에 정의되어 있기 때문에 start_decl () 함수와 finish_decl () 함수를 연속으로 실행함으로써 변수를 정의할 수 있기 때문이다. 물론 전역변수의 값이 start_decl () 함수에 전달되는 것은 당연하다. 이제 마지막 'c' 변수의 경우 옆에 초기화자가 붙어 있기 때문에 다른 변수와 약간 다른 수행을 거치게 된다. start_decl () 함수가 실행된 후 token 으로 '=' 를 만났을 때, start_init () 함수를 수행하게 되며, 모든 해석이 끝났을 때, finish_init () 함수와 finish_decl () 함수가 수행된다.

**(3)** char *a = "Hello world";

세번째 예제로써, 포인터 변수의 선언을 예로 들었다. 이 예제에서 봐야 할 부분은 '*' 를 처리하는 부분일 것이다. 실제로 '*' 가 올 경우 make_pointer_declarator () 함수에 의해서 INDIRECT_REF node 가 생성되고 이에 대해 declarator 가 start_decl () 함수에 의해서 생성되게 된다.

```
program -> extdefs
  extdefs -> extdef
    extdef -> datadef
      datadef -> declspecs_ts setspecs initdecls ';'
        declspecs_ts -> declspecs_nosc_ts_nosa_noea
          declspecs_nosc_ts_nosa_noea -> typespec_nonattr
            typespec_nonattr -> typespec_reserved_nonattr
              typespec_reserved_nonattr -> TYPESPEC
        setspecs
        initdecls -> initdcl
          initdcl -> declarator maybeasm maybe_attribute '=' init
            declarator -> notype_declarator
              notype_declarator -> '*' maybe_type_quals_attrs notype_declarator
                maybe_type_quals_attrs
                notype_declarator -> IDENTIFIER
            maybeasm
            maybe_attribute
          init -> expr_no_commas
            expr_no_commas -> cast_expr
              cast_expr -> unary_expr
                unary_expr -> primary
                  primary -> string
                    string -> STRING
```

**(4)** int abc[3] = { 1, 2, 3 };

네번째 예제로써, 다른 형태의 포인터 변수 선언을 예로 들었다. 위에서는 '*' 를 어떻게 처리하느냐에 대해서 살펴보았다면, 여기에서는 [...] 부분이 어떻게 처리되는지를 봐야 할 것이다. 변수를 선언하는 것이기 때문에, start_decl () 등등의 함수들이 호출되며, 초기화자가 존재하기 때문에 start_init () 함수 또한 호출이 되어 질 것이다. 여기서 주의 깊게 되야 할 부분은 declarator 라벨인데, notype_declarator 라벨에서 얻은 변수의 이름과, array_declarator 에서 얻은 [...] 내의 구성 사이를 set_array_declarator_type () 함수를 통해서 연결해 주게 된다. array_declarator 라벨에서는 build_array_declarator () 함수를 통해서 ARRAY_REF tree node 를 생성하게 된다.

```
datadef -> declspecs_ts setspecs initdecls ';'
  declspecs_ts -> declspecs_nosc_ts_nosa_noea
    declspecs_nosc_ts_nosa_noea -> typespec_nonattr
      typespec_nonattr -> typespec_reserved_nonattr
        typespec_reserved_nonattr -> TYPESPEC
  setspecs
  initdecls -> initdcl
    initdcl -> declarator maybeasm maybe_attribute '=' init
      declarator -> notype_declarator
        notype_declarator -> notype_declarator array_declarator
          notype_declarator -> IDENTIFIER
          array_declarator -> '[' expr ']'
            expr -> nonnull_exprlist
              nonnull_exprlist -> expr_no_commas
                cast_expr -> unary_expr
                  unary_expr -> primary
                    primary -> CONSTANT
      maybeasm
      maybe_attribute
      init -> '{' initlist_maybe_comma '}'
        initlist_maybe_comma -> initlist1 maybecomma
          initlist1 -> initlist1 ',' initelt
            initlist1 -> initlist1 ',' initelt
              initlist1 -> initelt
                initelt -> initval
                  initval -> expr_no_commas
                    expr_no_commas -> cast_expr
                      cast_expr -> unary_expr
                        unary_expr -> primary
                          primary -> CONSTANT
              initelt -> initval
                initval -> expr_no_commas
                  expr_no_commas -> cast_expr
                    cast_expr -> unary_expr
                      unary_expr -> primary
                        primary -> CONSTANT
            initelt -> initval
              initval -> expr_no_commas
                expr_no_commas -> cast_expr
                  cast_expr -> unary_expr
                    unary_expr -> primary
                      primary -> CONSTANT

          maybecomma
```

**(5)** extern int __attribute__((unused)) nvec;

네번째 예제로는, __attribute__ 가 추가되었으며, extern 으로 선언된 변수를 가르킨다. 여기에서 살펴봐야 할 것은 __attribute__ 가 처음으로 등장하였으며, int 형 앞에 extern 이 왔다는 것일 것이다. extern 이 int 앞에 옮으로써 달라지는 것은 int 를 구성하던 TREE_LIST node 의 chain 에 register 에 관한 TREE_LIST node 가 온다는 것이다. 뒤에서는 __attribute__ ((unused)) 가 오게 되는데, 이에 대해서 어떻게 처리되는지 궁금할 수 있는데, 이도 다른 것과 마찬가지로 TREE_LIST node 를 처리된다.

```
program -> extdefs
  extdefs -> extdef
    extdef -> datadef
```

```
datadef -> declspecs_ts setspecs initdecls ';'
  declspecs_ts -> declspecs_sc_ts_nosa_ea
    declspecs_sc_ts_nosa_ea -> declspecs_sc_ts_nosa_noea attributes
      declspecs_sc_ts_nosa_noea -> declspecs_sc_nots_nosa_noea
                                   typespec_nonattr
        declspecs_sc_nots_nosa_noea -> SCSPEC
        typespec_nonattr -> typespec_reserved_nonattr
          typespec_reserved_nonattr -> TYPESPEC
      attributes -> ATTRIBUTE '(' '(' attribute_list ')' ')'
        attribute_list -> attrib
          attrib -> any_word
            any_word -> identifier
              identifier -> IDENTIFIER
  setspecs
  initdecls
```

하지만, attribute 들과 int 를 대표하는 node 가 TREE_LIST 의 purpose, value, chain 내의 위치는 다른데, 결과적으로 아래와 같은 TREE node 를 구성하게 된다.

```
<tree_list 0x401e53fc
   purpose <tree_list 0x401e53e8
       purpose <identifier_node 0x401f59c0 unused>>
   chain <tree_list 0x401e53d4 static
       value <identifier_node 0x401dd100 int tree_0
           global <type_decl 0x401daa80 int>
           rid 0x401dd100 "int">
       chain <tree_list 0x401e53c0
           value <identifier_node 0x401dd180 register tree_0
               rid 0x401dd180 "extern">>>>
```

위의 node 에서 볼 수 있듯이, attribute 의 경우 purpose 에 들어가고, extern 을 포함하는 int 는 chain 에 들어가게 된다. 이에 대해서 이렇게 언급하는 이유는 결과적으로 어떤 declarator 를 선언하기 위해서는 전역 변수 current_declspecs 와 prefix_attributes, all_prefix_attributes 에 각기 이것들이 분리되어 들어가야 되는데, 이 분리의 역활을 담당하는 것이 split_specs_attrs () 함수가 하게 된다. 그럼 실제로 분리가 이루어진 후의 모습은 아래와 같다.

current_declspecs 는 아래와 같은 모습이다.

```
<tree_list 0x401e53d4 static
   value <identifier_node 0x401dd100 int tree_0
       global <type_decl 0x401daa80 int type <integer_type 0x401da380 int>
           VOID file <built-in> line 0
           align 1>
       rid 0x401dd100 "int">
   chain <tree_list 0x401e53c0
       value <identifier_node 0x401dd180 register tree_0
           rid 0x401dd180 "register">>>
```

prefix_attributes 는 아래와 같은 모습이다.

```
<tree_list 0x401e53e8
   purpose <identifier_node 0x401f59c0 unused>>
```

이렇게 위와 같이 분리된 형태로 start_decl () 함수가 호출되어지게 된다.

**(6)** register int eax_ __asm__("eax");

이 예제의 경우, 끝에 __asm__ 구문이 붙어 있다는 것이 다른 부분일 것이다. 이 구문을 처리하는 것이 maybeasm 라벨에서 이루어지며, 여기서 생성된 STRING tree node 는 start_decl () 함수에게는 건내지지 않고, start_init () 함수와 finish_decl () 함수의 인자로 건내지게 된다.

```
decl -> declspecs_ts setspecs initdecls ';'
  declspecs_ts -> declspecs_sc_ts_nosa_noea
    declspecs_sc_ts_nosa_noea -> declspecs_sc_nots_nosa_noea typespec_nonattr
      declspecs_sc_nots_nosa_noea -> SCSPEC
      typespec_nonattr -> typespec_reserved_nonattr
        typespec_reserved_nonattr -> TYPESPEC
  setspecs
  initdecls -> initdcl
    initdcl -> declarator maybeasm maybe_attribute
      declarator -> notype_declarator
        notype_declarator -> IDENTIFIER
      maybeasm -> ASM_KEYWORD '(' string ')'
        string -> STRING
      maybe_attribute
```

**(7)** struct abc { char * a; } __attribute__((packed));

이제는 구조체에 대해 알아보도록 하자. 살펴봐야 할 부분이 있다면, structsp_attr 라벨과, component_decl_list 라벨을 들 수 있을 것이다. 구조체의 경우 RECORD_TYPE node 를 사용하게 되는데, structsp_attr 라벨에서 이 node 를 생성하는 start_struct () 함수와 finish_struct () 함수가 호출되며, { 와 } 에 둘러싸여 구조체 내부를 구성하는 변수들은 grokfield () 함수와 decl_attributes () 함수에 의해서 FIELD_DECL node 가 생성되게 된다.

```
program -> extdefs
  extdefs -> extdef
    extdef -> datadef
      datadef -> declspecs ';'
        declspecs -> declspecs_nosc_ts_nosa_ea
          declspecs_nosc_ts_nosa_ea -> typespec_attr
            typespec_attr -> typespec_reserved_attr
              typespec_reserved_attr -> structsp_attr
                structsp_attr -> struct_head identifier
                                 '{' component_decl_list '}' maybe_attribute
                  struct_head -> STRUCT
                  identifier -> IDENTIFIER
                  component_decl_list -> component_decl_list2
                    component_decl_list2 -> component_decl_list2
                                            component_decl ';'
                      component_decl_list2
                      component_decl -> declspecs_nosc_ts setspecs components
                        declspecs_nosc_ts -> declspecs_nosc_ts_nosa_noea
                          declspecs_nosc_ts_nosa_noea -> typespec_nonattr
                            typespec_nonattr -> typespec_reserved_nonattr
                              typespec_reserved_nonattr -> TYPESPEC
                        setspecs
                        components -> component_declarator
                          component_declarator -> save_filename save_lineno
                                                  declarator maybe_attribute
                            save_filename
                            save_lineno
                            declarator -> notype_declarator
```

```
                    notype_declarator -> '*' maybe_type_quals_attrs
                                         notype_declarator
                      maybe_type_quals_attrs
                      notype_declarator -> IDENTIFIER
                  maybe_attribute
              maybe_attribute -> attributes
                attributes -> attribute
                  attribute -> ATTRIBUTE '(' '(' attribute_list ')' ')'
                    attribute_list -> attrib
                      attrib -> any_word
                        any_word -> identifier
                          identifier -> IDENTIFIER
```

**(8)** union abc { float f; };

UNION 의 경우, STRUCT 와 해석하는 과정에서의 다른 점은 찾기 힘들다. STRUCT 가 tree node 를 구성하는 과정에서 start_struct () 함수를 통해 RECORD_TYPE tree node 를 만들었다면, UNION 은 start_struct () 함수를 이용하여 UNION_TYPE 을 만들게 된다.

```
program -> extdefs
  extdefs -> extdef
    extdef -> datadef
      datadef -> declspecs ';'
        declspecs -> declspecs_nosc_ts_nosa_ea
          declspecs_nosc_ts_nosa_ea -> typespec_attr
            typespec_attr -> typespec_reserved_attr
              typespec_reserved_attr -> structsp_attr
                structsp_attr -> union_head identifier '{'
                                 component_decl_list '}'
                                 maybe_attribute
                  union_head -> UNION
                  identifier -> IDENTIFIER
                  component_decl_list -> component_decl_list2
                    component_decl_list2 -> component_decl_list2
                                            component_decl ';'
                      component_decl_list2
                      component_decl -> declspecs_nosc_ts setspecs
                                        components
                        declspecs_nosc_ts -> declspecs_nosc_ts_nosa_noea
                          declspecs_nosc_ts_nosa_noea -> typespec_nonattr
                            typespec_nonattr -> typespec_reserved_nonattr
                              typespec_reserved_nonattr -> TYPESPEC
                         setspecs
                        components -> component_declarator
                          component_declarator -> save_filename save_lineno
                                                  declarator
                                                  maybe_attribute
                            save_filename
                            save_lineno
                            declarator -> notype_declarator
                              notype_declarator -> IDENTIFIER
                            maybe_attribute
                  maybe_attribute
```

**(9)** enum abc { HELLO = 0, WORLD };

ENUM 의 경우, Tree node 를 구성하기 위해서, start_enum () 함수와 finish_enum () 함수를 각각 사용하게 되며, ENUM 의 내부를 구성하는 요소들은 build_enumerator () 함수에 의해서 구성되게 된다. 아래의 yacc debug 결과는 앞단에 있는 program 라벨 부분을 생략한 것이다.

```
datadef -> declspecs ';'
  declspecs -> declspecs_nosc_ts_nosa_ea
    declspecs_nosc_ts_nosa_ea -> typespec_attr
      typespec_attr -> typespec_reserved_attr
        typespec_reserved_attr -> structsp_attr
          structsp_attr -> enum_head identifier '{' enumlist maybecomma_warn
                           '}' maybe_attribute
            enum_head -> ENUM
            identifier -> IDENTIFIER
            enumlist -> enumlist ',' enumerator
              enumlist -> enumerator
                enumerator -> identifier '=' expr_no_commas
                  identifier -> IDENTIFIER
                  expr_no_commas -> cast_expr
                    cast_expr -> unary_expr
                      unary_expr -> primary
                        primary -> CONSTANT
              enumerator -> identifier
                identifier -> IDENTIFIER
            maybecomma_warn
            maybe_attribute
```

## 제 4 절 Expression

위의 섹션에서 우리는 Data Type 에 대해서 설명을 하였다. 즉 어떤 전역 변수를 선언할 때 혹은 구조체, 공용체 등등의 여러 type 들을 선언하는 것을 총괄하여 처리한다고 말할 수 있을 것이다. 이 섹션에서는 이제 위에서 언급했던 "초기화자" 내에서 표현되어질 수 있는 표현식에 대해 알아 보도록 하겠다.

표현식이라고 하면 여러 가지 종류의 표현을 포함할 수 있을 텐데, 예를 들다면 아래와 같은 종류가 있다고 할 수 있을 것이다.

- if, for, while, ... 등등의 여러 statement 표현식
- label 과 같은 표현식
- +, −, ∗, / 와 같은 사칙 연산 및, &, |, ... 등등과 같은 BIT 연산자들 등등의 표현식
- 문자열 "..." 형태의 표현식

위의 언급한 종류를 제외하고 다른 여러 표현식 또한 포함되어질 수 있을 것이다. 이러한 분류를 크게 생각한다면, 네 가지로 GCC 에서는 요약을 하고 있는데, 함수를 제외한 것은 아래와 같다.

- Operator
- Statement
- Declarator
- Label

Operator 의 경우 여러 사칙 연산 및 연산자관련 표현을 말하며 Statement 의 경우 if, for, while, ... 과 같은 여러 표현식을 말한다. Declarator 의 경우 위의 섹션에서 언급한 data type 을 총괄하여 말한다고 할 수 있을 것이다. Label 의 경우 goto 문이 이용하기 위한 정보라고 말 할 수 있다. 각각에 대한 표현식에 대해 이제부터 하위 섹션에서 언급해 보자.

## 4.1 Operator 표현식

이제 Operator 표현식에 대한 yacc 문법을 아래에서 보게 될 것이다. 물론 사칙 연산을 제외한 여러 연산자도 아래에 있지만, 다른 표현식 또한 존재하는다. 대표적으로는 Cast 표현식을 들수 있다. 대부분의 연산 변수 앞에 (...) 와 같은 형식으로 보일 수 있는 형태를 cast_expr 라벨에 언급하고 있음을 알 수 있을 것이다.

```
expr: nonnull_exprlist
        { $$ = build_compound_expr ($1); }
    ;

exprlist:
      /* empty */
        { $$ = NULL_TREE; }
    | nonnull_exprlist
    ;

nonnull_exprlist:
    expr_no_commas
        { $$ = build_tree_list (NULL_TREE, $1); }
    | nonnull_exprlist ',' expr_no_commas
        { chainon ($1, build_tree_list (NULL_TREE, $3)); }
    ;

expr_no_commas:
      cast_expr
    | expr_no_commas '+' expr_no_commas
        { $$ = parser_build_binary_op ($2, $1, $3); }
    | expr_no_commas '-' expr_no_commas
        { $$ = parser_build_binary_op ($2, $1, $3); }
    | expr_no_commas '*' expr_no_commas
        { $$ = parser_build_binary_op ($2, $1, $3); }
    | expr_no_commas '/' expr_no_commas
        { $$ = parser_build_binary_op ($2, $1, $3); }
    | expr_no_commas '%' expr_no_commas
        { $$ = parser_build_binary_op ($2, $1, $3); }
    | expr_no_commas LSHIFT expr_no_commas
        { $$ = parser_build_binary_op ($2, $1, $3); }
    | expr_no_commas RSHIFT expr_no_commas
        { $$ = parser_build_binary_op ($2, $1, $3); }
    | expr_no_commas ARITHCOMPARE expr_no_commas
        { $$ = parser_build_binary_op ($2, $1, $3); }
    | expr_no_commas EQCOMPARE expr_no_commas
        { $$ = parser_build_binary_op ($2, $1, $3); }
    | expr_no_commas '&' expr_no_commas
        { $$ = parser_build_binary_op ($2, $1, $3); }
    | expr_no_commas '|' expr_no_commas
        { $$ = parser_build_binary_op ($2, $1, $3); }
    | expr_no_commas '^' expr_no_commas
        { $$ = parser_build_binary_op ($2, $1, $3); }
    | expr_no_commas ANDAND
        { $1 = truthvalue_conversion (default_conversion ($1));
          skip_evaluation += $1 == boolean_false_node; }
      expr_no_commas
        { skip_evaluation -= $1 == boolean_false_node;
          $$ = parser_build_binary_op (TRUTH_ANDIF_EXPR, $1, $4); }
    | expr_no_commas OROR
```

```
                { $1 = truthvalue_conversion (default_conversion ($1));
                  skip_evaluation += $1 == boolean_true_node; }
          expr_no_commas
                { skip_evaluation -= $1 == boolean_true_node;
                  $$ = parser_build_binary_op (TRUTH_ORIF_EXPR, $1, $4); }
        | expr_no_commas '?'
                { $1 = truthvalue_conversion (default_conversion ($1));
                  skip_evaluation += $1 == boolean_false_node; }
          expr ':'
                { skip_evaluation += (($1 == boolean_true_node)
                                      - ($1 == boolean_false_node)); }
          expr_no_commas
                { skip_evaluation -= $1 == boolean_true_node;
                  $$ = build_conditional_expr ($1, $4, $7); }
        | expr_no_commas '?'
                { if (pedantic)
                    pedwarn ("ISO C forbids omitting the middle term of a ?: expression");
                  /* 첫번째 operand 가 오직 한번만 계산되도록 확실히 한다.                    */
                  $<ttype>2 = save_expr ($1);
                  $1 = truthvalue_conversion (default_conversion ($<ttype>2));
                  skip_evaluation += $1 == boolean_true_node; }
          ':' expr_no_commas
                { skip_evaluation -= $1 == boolean_true_node;
                  $$ = build_conditional_expr ($1, $<ttype>2, $5); }
        | expr_no_commas '=' expr_no_commas
                { char class;
                  $$ = build_modify_expr ($1, NOP_EXPR, $3);
                  class = TREE_CODE_CLASS (TREE_CODE ($$));
                  if (IS_EXPR_CODE_CLASS (class))
                    C_SET_EXP_ORIGINAL_CODE ($$, MODIFY_EXPR);
                }
        | expr_no_commas ASSIGN expr_no_commas
                { char class;
                  $$ = build_modify_expr ($1, $2, $3);
                  /* 이것은 truthvalue_conversion 내 경고들을 금한다.          */
                  class = TREE_CODE_CLASS (TREE_CODE ($$));
                  if (IS_EXPR_CODE_CLASS (class))
                    C_SET_EXP_ORIGINAL_CODE ($$, ERROR_MARK);
                }
        ;

cast_expr:
        unary_expr
        | '(' typename ')' cast_expr %prec UNARY
                { $$ = c_cast_expr ($2, $4); }
        ;

unary_expr:
        primary
        | '*' cast_expr %prec UNARY
                { $$ = build_indirect_ref ($2, "unary *"); }
        /* __extension__ 는 다음 primary 를 위해 -pedantic 비활성화한다.          */
        | extension cast_expr %prec UNARY
                { $$ = $2;
                  RESTORE_WARN_FLAGS ($1); }
        | unop cast_expr %prec UNARY
                { $$ = build_unary_op ($1, $2, 0);
```

```
            overflow_warning ($$); }
      /* 포인터 또는 벌s label 의 address 를 참조한다.        */
      | ANDAND identifier
            { $$ = finish_label_address_expr ($2); }
      | sizeof unary_expr %prec UNARY
            { skip_evaluation--;
              if (TREE_CODE ($2) == COMPONENT_REF
                  && DECL_C_BIT_FIELD (TREE_OPERAND ($2, 1)))
                error ("'sizeof' applied to a bit-field");
              $$ = c_sizeof (TREE_TYPE ($2)); }
      | sizeof '(' typename ')' %prec HYPERUNARY
            { skip_evaluation--;
              $$ = c_sizeof (groktypename ($3)); }
      | alignof unary_expr %prec UNARY
            { skip_evaluation--;
              $$ = c_alignof_expr ($2); }
      | alignof '(' typename ')' %prec HYPERUNARY
            { skip_evaluation--;
              $$ = c_alignof (groktypename ($3)); }
      | REALPART cast_expr %prec UNARY
            { $$ = build_unary_op (REALPART_EXPR, $2, 0); }
      | IMAGPART cast_expr %prec UNARY
            { $$ = build_unary_op (IMAGPART_EXPR, $2, 0); }
      ;
primary:
      IDENTIFIER
            {
              if (yychar == YYEMPTY)
                yychar = YYLEX;
              $$ = build_external_ref ($1, yychar == '(');
            }
      | CONSTANT
      | string
            { $$ = combine_strings ($1); }
      | VAR_FUNC_NAME
            { $$ = fname_decl (C_RID_CODE ($$), $$); }
      | '(' typename ')' '{'
            { start_init (NULL_TREE, NULL, 0);
              $2 = groktypename ($2);
              really_start_incremental_init ($2); }
        initlist_maybe_comma '}' %prec UNARY
            { tree constructor = pop_init_level (0);
              tree type = $2;
              finish_init ();

              if (pedantic && ! flag_isoc99)
                pedwarn ("ISO C89 forbids compound literals");
              $$ = build_compound_literal (type, constructor);
            }
      | '(' expr ')'
            { char class = TREE_CODE_CLASS (TREE_CODE ($2));
              if (IS_EXPR_CODE_CLASS (class))
                C_SET_EXP_ORIGINAL_CODE ($2, ERROR_MARK);
              $$ = $2; }
      | '(' error ')'
            { $$ = error_mark_node; }
      | compstmt_primary_start compstmt_nostart ')'
```

```
165             { tree saved_last_tree;

                 if (pedantic)
                   pedwarn ("ISO C forbids braced-groups within expressions");
                pop_label_level ();
170
                saved_last_tree = COMPOUND_BODY ($1);
                RECHAIN_STMTS ($1, COMPOUND_BODY ($1));
                last_tree = saved_last_tree;
                TREE_CHAIN (last_tree) = NULL_TREE;
175             if (!last_expr_type)
                  last_expr_type = void_type_node;
                $$ = build1 (STMT_EXPR, last_expr_type, $1);
                TREE_SIDE_EFFECTS ($$) = 1;
              }
180         | compstmt_primary_start error ')'
              {
                pop_label_level ();
                last_tree = COMPOUND_BODY ($1);
                TREE_CHAIN (last_tree) = NULL_TREE;
185             $$ = error_mark_node;
              }
            | primary '(' exprlist ')' %prec '.'
              { $$ = build_function_call ($1, $3); }
            | VA_ARG '(' expr_no_commas ',' typename ')'
190           { $$ = build_va_arg ($3, groktypename ($5)); }

            | CHOOSE_EXPR '(' expr_no_commas ',' expr_no_commas ',' expr_no_commas ')'
              {
                tree c;
195
                c = fold ($3);
                STRIP_NOPS (c);
                if (TREE_CODE (c) != INTEGER_CST)
                  error ("first argument to __builtin_choose_expr not a constant");
200             $$ = integer_zerop (c) ? $7 : $5;
              }
            | TYPES_COMPATIBLE_P '(' typename ',' typename ')'
              {
                tree e1, e2;
205
                e1 = TYPE_MAIN_VARIANT (groktypename ($3));
                e2 = TYPE_MAIN_VARIANT (groktypename ($5));

                $$ = comptypes (e1, e2)
210               ? build_int_2 (1, 0) : build_int_2 (0, 0);
              }
            | primary '[' expr ']' %prec '.'
              { $$ = build_array_ref ($1, $3); }
            | primary '.' identifier
215           {
                $$ = build_component_ref ($1, $3);
              }
            | primary POINTSAT identifier
              {
220             tree expr = build_indirect_ref ($1, "->");
```

```
                    $$ = build_component_ref (expr, $3);
                }
        | primary PLUSPLUS
                { $$ = build_unary_op (POSTINCREMENT_EXPR, $1, 0); }
        | primary MINUSMINUS
                { $$ = build_unary_op (POSTDECREMENT_EXPR, $1, 0); }
        ;

unop:   '&'
                { $$ = ADDR_EXPR; }
        | '-'
                { $$ = NEGATE_EXPR; }
        | '+'
                { $$ = CONVERT_EXPR;
  if (warn_traditional && !in_system_header)
    warning ("traditional C rejects the unary plus operator");
                }
        | PLUSPLUS
                { $$ = PREINCREMENT_EXPR; }
        | MINUSMINUS
                { $$ = PREDECREMENT_EXPR; }
        | '~'
                { $$ = BIT_NOT_EXPR; }
        | '!'
                { $$ = TRUTH_NOT_EXPR; }
        ;

sizeof:
       SIZEOF { skip_evaluation++; }
        ;

alignof:
       ALIGNOF { skip_evaluation++; }
        ;

typename:
        declspecs_nosc
                { pending_xref_error ();
                  $<ttype>$ = $1; }
        absdcl
                { $$ = build_tree_list ($<ttype>2, $3); }
        ;

/* STRING_CST 들로 묶여져 있는 STRING_CST 를 생산한다.          */
string:
        STRING
      | string STRING
                {
                  static int last_lineno = 0;
                  static const char *last_input_filename = 0;
                  $$ = chainon ($1, $2);
                  if (warn_traditional && !in_system_header
                      && (lineno != last_lineno || !last_input_filename ||
                          strcmp (last_input_filename, input_filename)))
                    {
                      warning ("traditional C rejects string concatenation");
                      last_lineno = lineno;
```

```
                    last_input_filename = input_filename;
280                 }
                }
        ;

compstmt_primary_start:
        '(' '{'
                { if (current_function_decl == 0)
                    {
                    error ("braced-group within expression allowed only inside a function");
                    YYERROR;
                    }
                  /* 우리는 이 level 에서 만약 나중에 확장되지 않을 것
                     같을 경우, 반드시 BLOCK 을 강제해야하며, block 들
                     내부에 포함되어 있는 전체 subtree 를 비활성화시키는
                     방법이 존재한다.          */
                  keep_next_level ();
                  push_label_level ();
                  compstmt_count++;
                  $$ = add_stmt (build_stmt (COMPOUND_STMT, last_tree));
                }
        ;

compstmt_nostart: '}'
                { $$ = convert (void_type_node, integer_zero_node); }
        | pushlevel maybe_label_decls compstmt_contents_nonempty '}' poplevel
                { $$ = poplevel (kept_level_p (), 1, 0);
                  SCOPE_STMT_BLOCK (TREE_PURPOSE ($5))
                    = SCOPE_STMT_BLOCK (TREE_VALUE ($5))
                    = $$; }
        ;

pushlevel: /* empty */
                { pushlevel (0);
                  clear_last_expr ();
                  add_scope_stmt (/*begin_p=*/1, /*partial_p=*/0);
                }
        ;

/* 0 을 읽거나 nested function 들이 jump 할 수 있는 label 들을 위한
   추가적인 forward-declaration 들을 읽는다.        */
maybe_label_decls:
          /* empty */
        | label_decls
                { if (pedantic)
                    pedwarn ("ISO C forbids label declarations"); }
        ;

label_decls:
          label_decl
        | label_decls label_decl
        ;

label_decl:
          LABEL identifiers_or_typenames ';'
                { tree link;
                  for (link = $2; link; link = TREE_CHAIN (link))
```

```
                {
                  tree label = shadow_label (TREE_VALUE (link));
                  C_DECLARED_LABEL_FLAG (label) = 1;
                  add_decl_stmt (label);
                }
            }
        ;

/* Typename 들을 포함하여, identifier 들의 nonempty list.     */
identifiers_or_typenames:
        identifier
                { $$ = build_tree_list (NULL_TREE, $1); }
        | identifiers_or_typenames ',' identifier
                { $$ = chainon ($1, build_tree_list (NULL_TREE, $3)); }
        ;

compstmt_contents_nonempty:
          stmts_and_decls
        | error
        ;

/* compound statement 의 몸체를 형성할 수 있는 declaration 들과 statement
   들의 (몇몇 label 뒤에 올 가능성이 있는) nonempty 시퀴즈들.
   NOTE: 우리는 declaration 들상에서 label 은 허용하지 않는다; 이것이 natural
   extension 으로 보일수 있겠지만, label 상에서의 attribute 들과 declaration
   상의 prefix attribute 들간에 충돌을 일으킨다.             */

stmts_and_decls:
          lineno_stmt_decl_or_labels_ending_stmt
        | lineno_stmt_decl_or_labels_ending_decl
        | lineno_stmt_decl_or_labels_ending_label
                {
                  pedwarn ("deprecated use of label at end of compound statement");
                }
        | lineno_stmt_decl_or_labels_ending_error
        ;

lineno_stmt_decl_or_labels_ending_stmt:
          lineno_stmt
        | lineno_stmt_decl_or_labels_ending_stmt lineno_stmt
        | lineno_stmt_decl_or_labels_ending_decl lineno_stmt
        | lineno_stmt_decl_or_labels_ending_label lineno_stmt
        | lineno_stmt_decl_or_labels_ending_error lineno_stmt
        ;

lineno_stmt_decl_or_labels_ending_decl:
          lineno_decl
        | lineno_stmt_decl_or_labels_ending_stmt lineno_decl
                { if (pedantic && !flag_isoc99)
                    pedwarn ("ISO C89 forbids mixed declarations and code"); }
        | lineno_stmt_decl_or_labels_ending_decl lineno_decl
        | lineno_stmt_decl_or_labels_ending_error lineno_decl
        ;

lineno_stmt_decl_or_labels_ending_label:
          lineno_label
        | lineno_stmt_decl_or_labels_ending_stmt lineno_label
```

```
        | lineno_stmt_decl_or_labels_ending_decl lineno_label
        | lineno_stmt_decl_or_labels_ending_label lineno_label
        | lineno_stmt_decl_or_labels_ending_error lineno_label
        ;

lineno_stmt_decl_or_labels_ending_error:
        errstmt
        | lineno_stmt_decl_or_labels errstmt
        ;

lineno_stmt_decl_or_labels:
          lineno_stmt_decl_or_labels_ending_stmt
        | lineno_stmt_decl_or_labels_ending_decl
        | lineno_stmt_decl_or_labels_ending_label
        | lineno_stmt_decl_or_labels_ending_error
        ;

errstmt: error ';'
        ;

lineno_stmt:
          save_filename save_lineno stmt
               { if ($3)
                  {
                    STMT_LINENO ($3) = $2;
                    /* ??? 우리는 현재 statement 에 대한 filename 을 기록할
                       방법을 가지고 있지않다.              이것은 어떤 부분에서 작은 문제
                       가 될 수 있지만, 이 문제가 tree level 에서 inlining 을
                       수행하는 동안 발생할 수 있을 것이다.                    */
                  }
               }
        ;

lineno_label:
          save_filename save_lineno label
               { if ($3)
                  {
                    STMT_LINENO ($3) = $2;
                  }
               }
        ;

/* Decl 전에 lineno 를 저장하는 이 조합은 decl 그 자체보다
   사용하기엔 일반적이다.              이것은 statement label 들이 허락된
   context 내에서 shift/reduce conflicts 를 피하기 위해서
   이다.    */
lineno_decl:
          save_filename save_lineno decl
               { }
        ;

poplevel: /* empty */
               { $$ = add_scope_stmt (/*begin_p=*/0, /*partial_p=*/0); }
        ;
```

이제부터는 실제 operator 표현식을 GCC 에서는 어떻게 처리하는지에 대해서 알아 보도록 하겠는데, 간단한 표현식부터 복잡한 표현식으로 전개해 나가겠다.

지금 우리가 여기에서 다루고 있는 것은 operator 이며, operator 가 나타날 수 있는 곳은 지정되어 있다고 할 수 있는데, operator 가 보일 수 있는 곳을 한번 언급해 보자.

1. 변수를 처음 선언하거나, 변수를 사용할 때, token '=' 뒤에 operator 표현식이 나타날 수 있다.
2. if, while, for 등등 (...) 로 둘러쌓이는 곳에서 나타날 수 있다.
3. ... ? ... : ... 와 같은 구문에 나타날 수 있다.
4. 함수를 호출할 때, 내부 parameter 의 값에 나타날 수 있다.
5. 배열을 선언할 때 [...] 사이에 operator 들이 나타날 수 있다.
6. 기타 등등

위에서 언급하였듯이, operator 표현식이 나타날 수 있는 곳은 상당히 많다. 우리가 알고자 하는 것은 이러한 operator 표현식을 어떻게 처리하는가에 대한 것이다. 아래에서 예제를 들어가며 설명을 하도록 하겠다.

**(1)** int i = 1 + 2;

첫번째 예제로써 간단한 "1 + 2" 연산을 어떻게 처리하는지에 대해서 보도록 하며, 앞으로 점점 복잡한 operator 표현식을 보도록 하자. 현재 operator 표현식이 token '=' 뒤에 나오기 때문에, 우리는 표현식을 처리할 label 이 init 라벨이라는 것을 알 수 있는데, 실제 init 라벨에서 "1 + 2"은 어떻게 처리되는지 살펴보도록 하자.

```
init -> expr_no_commas
  expr_no_commas -> expr_no_commas '+' expr_no_commas
    expr_no_commas -> cast_expr
      cast_expr -> unary_expr
        unary_expr -> primary
          primary -> CONSTANT
    expr_no_commas -> cast_expr
      cast_expr -> unary_expr
        unary_expr -> primary
          primary -> CONSTANT
```

이 부분에서 살펴봐야 할 부분이 있다면, expr_no_commas 라벨인데, 실제 $prefix/gcc/c-parse.in 파일을 보면 알겠지만, token '+', '−', '*', '/', '%', LSHIFT, RSHIFT, ARITHCOMPARE[1], EQCOMPARE, '&', '|', '^' 의 경우, 모두 parser_build_binary_op () 함수에 의해서 처리되어진다. ANDAND, OROR, '+=', '-=' 등등의 operator 의 경우 다르게 수행이 되는데, 그에 대해서는 아래에서 계속 볼 것이다.

**(2)** int i = 1 + 2 * 3;

이 예제는 위와 마찬가지로 모두 parser_build_binary_op () 함수에 의해서 처리되어진다.

```
init -> expr_no_commas
  expr_no_commas -> expr_no_commas '+' expr_no_commas
    expr_no_commas -> cast_expr
      cast_expr -> unary_expr
        unary_expr -> primary
          primary -> CONSTANT
    expr_no_commas -> expr_no_commas '*' expr_no_commas
```

[1] 이것은 >, < 와 같은 비교 operator 들을 가르킨다.

```
expr_no_commas -> cast_expr
  cast_expr -> unary_expr
    unary_expr -> primary
      primary -> CONSTANT
expr_no_commas -> cast_expr
  cast_expr -> unary_expr
    unary_expr -> primary
      primary -> CONSTANT
```

**(3)** int i = 0xff && 0xf0;

token ANDAND 와 ORROR 은 비록 마지막에는 parser_build_binary_op () 함수를 통해서, tree node 가 구성되지만, 중간에 truthvalue_conversion () 함수에 의해서 boolean_true_node 혹은 boolean_false_node 와 비교되어 진다. 이를 통해서 $prefix/gcc/c-common.c 파일에 선언되어 있는 전역 변수 skip_evaluation 에 반영이 되게 된다.

```
init -> expr_no_commas
  expr_no_commas -> expr_no_commas ANDAND expr_no_commas
    expr_no_commas -> cast_expr
      cast_expr -> unary_expr
        unary_expr -> primary
          primary -> CONSTANT
    expr_no_commas -> cast_expr
      cast_expr -> unary_expr
        unary_expr -> primary
          primary -> CONSTANT
```

**(4)** int i = 1 ? 2 : 3;

IF 구문을 대신할 수 있는 ... ? ... : ... 구문의 경우, operator 표현식에 포함이 되는데, 결과적으로는 build_conditional_expr () 함수에 의해서 tree node 가 구성되게 되며, token '?' 앞단에 놓여져 있는 표현식에 따라서 뒤의 실행 구문이 변경되게 되며, 표현식에 대한 true, false 는 truthvalue_conversion () 함수에 의해서 구성되게 된다.

**(5)** int i = a = 2;

이 구문의 경우, int i 가 선언되기 전에 a 라는 변수가 앞에서 선언되었다고 가정을 했을 때, a 의 변수를 수정한 후, 그에 대한 결과를 변수 i 에 넣게 되는데, 이에 대한 처리는 build_modify_expr () 함수에 의해서 처리가 되어진다.

**(6)** int i = a += 2;

이것은 위의 예제 **(5)** 와 마찬가지로 build_modify_expr () 함수에 의해서 처리가 되어지며, 위의 '+=' 뿐만 아니라, token ASSIGN 으로 분류되는 모든 것에 대해 처리하게 된다.

**(7)** int i = (1 + 2);

이 예제는 위의 **(1)** 예제와 같다고 할 수 있지만, (...) 로 둘러쌓여 있기 때문에, 조금 다른 의미를 가지게 된다. expr 라벨에서 build_compound_expr () 함수가 호출되게 되는데, 이 함수에서는 하위 라벨인 nonnull_exprlist 에서 건내받은 tree node 를 COMPOUND_EXPR tree node 를 이용하여 둘러싼게 된다. 하위 라벨인 nonnull_exprlist 라벨에서는 그것의 하위 라벨인 expr_no_commas 를 build_tree_list () 로 또한 감싸게 된다. '+' operator 로 인해 생성되는 것은 위의 예제에서 이미 언급하였다.

```
init -> expr_no_commas
  expr_no_commas -> cast_expr
    cast_expr -> unary_expr
```

```
unary_expr -> primary
  primary -> '(' expr ')'
    expr -> nonnull_exprlist
      nonnull_exprlist -> expr_no_commas
        expr_no_commas -> expr_no_commas '+' expr_no_commas
          expr_no_commas -> cast_expr
            cast_expr -> unary_expr
              unary_expr -> primary
                primary -> CONSTANT
          expr_no_commas -> cast_expr
            cast_expr -> unary_expr
              unary_expr -> primary
                primary -> CONSTANT
```

**(8)** int i = (int ) *a;

이 예제와 같은 cast 를 포함하는 표현식의 경우, 앞에 나오는 type 과 뒤에 나오는 identifier 를 묶기 위해서, c_cast_expr () 함수를 사용하게 된다. 그리고 뒤에 나오는 *a 를 표현식으로 묶기 위해서, build_indirect_ref () 함수를 사용하게 된다.

```
init -> expr_no_commas
  expr_no_commas -> cast_expr
    cast_expr -> '(' typename ')' cast_expr
      typename -> declspecs_nosc absdcl
        declspecs_nosc -> declspecs_nosc_ts_nosa_noea
          declspecs_nosc_ts_nosa_noea -> typespec_nonattr
            typespec_nonattr -> typespec_reserved_nonattr
              typespec_reserved_nonattr -> TYPESPEC
        absdcl
      cast_expr -> unary_expr
        unary_expr -> '*' cast_expr
          cast_expr -> unary_expr
            unary_expr -> unary_expr
              primary -> IDENTIFIER
```

**(9)** int i = sizeof (a);

예제의 sizeof (a) 에서 a 는 앞에서 미리 선언된 identifier 로 가정한다. sizeof 의 경우, GCC 에서의 처리는 c_sizeof () 함수에 의해서 이루어지게 된다. sizeof 의 경우, 여러가지로 표현될 수 있는데, 함수형태로 쓰일 수도 있고, 뒤에 단순히 문자가 올수 있다. 즉, unary_expr 라벨이 표현하는 모든 표현식이 뒤에 올수 있다고 할 수 있을 것이다.

```
init -> expr_no_commas
  expr_no_commas -> cast_expr
    cast_expr -> unary_expr
      unary_expr -> sizeof unary_expr
        sizeof -> SIZEOF
        unary_expr -> primary
          primary -> '(' expr ')'
            expr -> nonnull_exprlist
              nonnull_exprlist -> expr_no_commas
                expr_no_commas -> cast_expr
                  cast_expr -> unary_expr
                    unary_expr -> primary
                      primary -> IDENTIFIER
```

**(10)** int i = alignof (a);

위의 예제 sizeof 와 거의 비슷하며, 받아들이는 문법 또한 sizeof 와 거의 동일하다. sizeof 의 경우, c_sizeof () 함수에 의해서 처리되었지만, alignof 의 경우, c_alignof_expr () 함수 혹은 c_alignof () 함수에 의해서 처리되게 된다.

**(11)** int *i = &a; int i = +1; int i = -1; int i = ++a; int i = --a; int i = 1; int i = !1;

이러한 표현식의 경우, 모두, build_unary_op () 함수에 의해서 처리가 되어지게 되는데, 물론 +, -, ++ 등등에 각기 해당하는 ...._EXPR 가 할당되게 되며, unop 라벨에 그것에 따른 표현식을 결정하게 된다.

**(12)** int i = a++; int i = a--;

이 예제의 경우, 위의 (11) 예제와 마찬가지로 build_unary_op () 함수에 의해서 처리가 되며, '++' 의 경우, POSTINCREMENT_EXPR tree node 가 '--' 의 경우, POSTDECREMENT_EXPR tree node 가 생성되게 된다. 이러한 것을 처리하게 되는 라벨은 primary 라벨에서 처리가 이루어지게 된다.

**(13)** int i = a[1];

이 예제의 경우, 배열의 2 번째 요소를 변수 i 에 넣게 되는데, 뒤의 a 배열을 처리하는 부분은 primary 라벨에 정의되어 있다. a[1] 표현식은 build_array_ref () 함수에 의해서 처리가 이루어지게 된다.

**(14)** int i = a.b;

이것은 a 구성요소의 b 참조를 가르키는 예제로써, a.b 의 표현은 build_component_ref () 함수에 의해서 표현되어지게 된다.

**(15)** int i = a→b;

이 예제는 a 구성요소의 b 포인터 참조로써, build_indirect_ref () 함수를 통해서 구한 expr 를 build_component_ref () 함수를 통해서, 실제 a→b 를 표현하게 된다.

**(16)** int i = ({ int a = 1; a; })

Braced-group 표현식에 대해서 정의를 한것으로, ({...}) 형태로 내부에 여러가지 표현식을 담을 수 있으며, 결과적으로 해당 return 값을 변수 i 에 넣게 된다. GCC 에서 이것을 처리하는 방법은 ({ token 을 만났을 경우, keep_next_level () 함수와, push_label_level () 함수를 호출하고, COMPOUND_STMT tree node 를 만든 후 statement tree 에 추가하게 된다. 이제, 초기 설정이 완료되었다면, compstmt_nostart 라벨을 통해서, statement 를 처리하고, 그에 대한 COMPOUND_BODY 를 획득한후, 마지막에, STMT_EXPR tree node 를 구성한 후 완료하게 된다. 이에 대한 세부 설명은 다음 문서에서 하게 될 것이다.

**(17)** int i = abc (1);

이 표현식의 경우, abc 함수 호출에 대한 결과값을 i 에 대입한다는 말인데, 이러한 함수 표현식을 처리하는 부분 또한 primary 라벨에 정의되어 있으며, build_function_call () 함수에 의해서 적당한 tree node 가 만들어지게 된다.

## 4.2 Statement 표현식

Statement 표현식의 경우, C 언어에서 위치할 수 있는 곳은 여러곳이다. 물론 함수내에서 올수 있고, Nested function 에도, ({...}) 표현식 내부에서도 올 수 있다. 이 하위 섹션에서 언급하고 있는 것은 ({...}) 표현식 내부에 표현되어질 수 있는 statement 표현식에 대해서 언급하고자 한다. 물론 함수내 혹은 netsted function 내에서 선언되는 것과 크게 다르다고 할 수 없을 것이다.

```
/* 어떤 label 들을 포함하지 않고, 실제 단일 statement 을 해석한다.                    */
stmt:
        compstmt
             { stmt_count++; $$ = $1; }
      | expr ';'
             { stmt_count++;
               $$ = c_expand_expr_stmt ($1); }
      | c99_block_start select_or_iter_stmt c99_block_end
             { if (flag_isoc99)
                 RECHAIN_STMTS ($1, COMPOUND_BODY ($1));
               $$ = NULL_TREE; }
      | BREAK ';'
             { stmt_count++;
               $$ = add_stmt (build_break_stmt ()); }
      | CONTINUE ';'
             { stmt_count++;
               $$ = add_stmt (build_continue_stmt ()); }
      | RETURN ';'
             { stmt_count++;
               $$ = c_expand_return (NULL_TREE); }
      | RETURN expr ';'
             { stmt_count++;
               $$ = c_expand_return ($2); }
      | ASM_KEYWORD maybe_type_qual '(' expr ')' ';'
             { stmt_count++;
               $$ = simple_asm_stmt ($4); }
      /* 이것은 단순히 output operand 들만 가진 경우이다.                          */
      | ASM_KEYWORD maybe_type_qual '(' expr ':' asm_operands ')' ';'
             { stmt_count++;
               $$ = build_asm_stmt ($2, $4, $6, NULL_TREE, NULL_TREE); }
      /* 이것은 input operand 들 또한 가진 경우이다.                              */
      | ASM_KEYWORD maybe_type_qual '(' expr ':' asm_operands ':'
        asm_operands ')' ';'
             { stmt_count++;
               $$ = build_asm_stmt ($2, $4, $6, $8, NULL_TREE); }
      /* 이것은 clobbered register 들 또한 가진 경우이다.                         */
      | ASM_KEYWORD maybe_type_qual '(' expr ':' asm_operands ':'
        asm_operands ':' asm_clobbers ')' ';'
             { stmt_count++;
               $$ = build_asm_stmt ($2, $4, $6, $8, $10); }
      | GOTO identifier ';'
             { tree decl;
               stmt_count++;
               decl = lookup_label ($2);
               if (decl != 0)
                 {
                   TREE_USED (decl) = 1;
                   $$ = add_stmt (build_stmt (GOTO_STMT, decl));
                 }
               else
                 $$ = NULL_TREE;
             }
      | GOTO '*' expr ';'
             { if (pedantic)
                 pedwarn ("ISO C forbids `goto *expr;'");
               stmt_count++;
               $3 = convert (ptr_type_node, $3);
```

```
                $$ = add_stmt (build_stmt (GOTO_STMT, $3)); }
        | ';'
                { $$ = NULL_TREE; }
        ;

compstmt: compstmt_start compstmt_nostart
                { RECHAIN_STMTS ($1, COMPOUND_BODY ($1));
                  last_expr_type = NULL_TREE;
                  $$ = $1; }
        ;

compstmt_start: '{' { compstmt_count++;
                      $$ = c_begin_compound_stmt (); }
        ;

compstmt_contents_nonempty:
          stmts_and_decls
        | error
        ;

/* C99 의 새 scope 들을 위해 생성된 시작과 끝 block 들.                */
c99_block_start: /* empty */
                { if (flag_isoc99)
                    {
                      $$ = c_begin_compound_stmt ();
                      pushlevel (0);
                      clear_last_expr ();
                      add_scope_stmt (/*begin_p=*/1, /*partial_p=*/0);
                    }
                  else
                    $$ = NULL_TREE;
                }
        ;

/* c99_block_start 와 c99_block_end 를 사용하는 production 들은 compstmt 내
   RECHAIN_STMTS ($1, COMPOUND_BODY ($1)); $$ = $2; 를 필요로 할 것이다.
   $1 은 c99_block_start 의 값과 c99_block_end 의 $2 값이다.      */
c99_block_end: /* empty */
                { if (flag_isoc99)
                    {
                      tree scope_stmt = add_scope_stmt (/*begin_p=*/0, /*partial_p=*/0);
                      $$ = poplevel (kept_level_p (), 0, 0);
                      SCOPE_STMT_BLOCK (TREE_PURPOSE (scope_stmt))
                        = SCOPE_STMT_BLOCK (TREE_VALUE (scope_stmt))
                        = $$;
                    }
                  else
                    $$ = NULL_TREE; }
        ;

/* lineno_labeled_stmt 와 비슷하지만, C99 내에서의 block 을 의미.            */
c99_block_lineno_labeled_stmt:
          c99_block_start lineno_labeled_stmt c99_block_end
                { if (flag_isoc99)
                    RECHAIN_STMTS ($1, COMPOUND_BODY ($1)); }
        ;
```

```
115 lineno_labeled_stmt:
        lineno_stmt
      | lineno_label lineno_labeled_stmt
      ;

120 select_or_iter_stmt:
        simple_if ELSE
            { c_expand_start_else ();
              $<itype>1 = stmt_count; }
        c99_block_lineno_labeled_stmt
125         { c_finish_else ();
              c_expand_end_cond ();
              if (extra_warnings && stmt_count == $<itype>1)
                warning ("empty body in an else-statement"); }
      | simple_if %prec IF
130         { c_expand_end_cond ();
              /* 이 경고가 simple_if 내 대신 여기 존재하는데, 그것은 empty if 가
                 else statement 에 따를 경우 발생하는 경고를 우리가 원하지 않기
                 때문이다.       증가 stmt_count 로 인해 우리는 만약 이것인 nested 'if'
                 일 경우 두번째 error 를 우지 않는다.                */
135           if (extra_warnings && stmt_count++ == $<itype>1)
                warning_with_file_and_line (if_stmt_file, if_stmt_line,
                                            "empty body in an if-statement"); }
      /* c_expand_end_cond 가 각 call 마다 c_expand_start_cond
         에 대응하여 오직 한번만 실행하도록 하라.                    그렇지 않을 경우
140      crash 가 발생한다.       */
      | simple_if ELSE error
            { c_expand_end_cond (); }
      /* 우리는 반드시 WHILE_STMT 의 condition 을 해석하기 전에
         WHILE_STMT node 를 build 해야 한다.    그래서 STMT_LINENO
145      는 ''while'' 를 포함하는 줄을 가르키지, close-parenthesis
         를 포함하는 줄을 가르키지 않는다.

         c_begin_while_stmt 는 WHILE_STMT node 를 반환하는데,
         우리는 나중에 condition 과 다른 토막들을 채울 수 있도록
150      c_finish_while_stmt_cond 에게 전한다.   */
      | WHILE
            { stmt_count++;
              $<ttype>$ = c_begin_while_stmt (); }
        '(' expr ')'
155         { $4 = truthvalue_conversion ($4);
              c_finish_while_stmt_cond (truthvalue_conversion ($4),
                                        $<ttype>2);
              $<ttype>$ = add_stmt ($<ttype>2); }
        c99_block_lineno_labeled_stmt
160         { RECHAIN_STMTS ($<ttype>6, WHILE_BODY ($<ttype>6)); }
      | do_stmt_start
        '(' expr ')' ';'
            { DO_COND ($1) = truthvalue_conversion ($3); }
      | do_stmt_start error
165         { }
      | FOR
            { $<ttype>$ = build_stmt (FOR_STMT, NULL_TREE, NULL_TREE,
                                      NULL_TREE, NULL_TREE);
              add_stmt ($<ttype>$); }
170     '(' for_init_stmt
            { stmt_count++;
```

```
                RECHAIN_STMTS ($<ttype>2, FOR_INIT_STMT ($<ttype>2)); }
        xexpr ';'
             { if ($6)
                 FOR_COND ($<ttype>2) = truthvalue_conversion ($6); }
        xexpr ')'
             { FOR_EXPR ($<ttype>2) = $9; }
        c99_block_lineno_labeled_stmt
             { RECHAIN_STMTS ($<ttype>2, FOR_BODY ($<ttype>2)); }
      | SWITCH '(' expr ')'
             { stmt_count++;
               $<ttype>$ = c_start_case ($3); }
        c99_block_lineno_labeled_stmt
             { c_finish_case (); }
      ;

/* 이것은 asm (``addextend %2,%1'': ``=dm'' (x), ``0'' (y), ``g'' (*x)) 에서
   첫번째 문자열과 colon 의의 다른 operand 들이다.                */
asm_operands: /* empty */
             { $$ = NULL_TREE; }
      | nonnull_asm_operands
      ;

nonnull_asm_operands:
        asm_operand
      | nonnull_asm_operands ',' asm_operand
             { $$ = chainon ($1, $3); }
      ;

asm_operand:
        STRING '(' expr ')'
             { $$ = build_tree_list (build_tree_list (NULL_TREE, $1), $3); }
      | '[' identifier ']' STRING '(' expr ')'
             { $$ = build_tree_list (build_tree_list ($2, $4), $6); }
      ;

asm_clobbers:
        string
             { $$ = tree_cons (NULL_TREE, combine_strings ($1), NULL_TREE); }
      | asm_clobbers ',' string
             { $$ = tree_cons (NULL_TREE, combine_strings ($3), $1); }
      ;

/* 같은 closeparen 의 number 가지 센 statement 들의 number 이다.       */
simple_if:
        if_prefix c99_block_lineno_labeled_stmt
             { c_finish_then (); }
     /* c_expand_end_cond 가 각 call 마다 c_expand_start_cond
        에 대응하여 오직 한번만 실행하도록 하라.                    그렇지 않을 경우
        crash 가 발생한다.       */
      | if_prefix error
      ;

if_prefix:
        /* 우리는 반드시 IF_STMT 의 condition 을 해석하기 전에
           IF_STMT node 를 build 해야 한다.    그래서 STMT_LINENO
           는 ``if'' 를 포함하는 줄을 가르키지, close-parenthesis
           를 포함하는 줄을 가르키지 않는다.
```

```
230             c_begin_if_stmt 는 IF_STMT node 를 반환하는데,
                우리는 나중에 condition 과 다른 토막들을 채울 수 있도록
                c_expand_start_cond 에게 전한다.    */
        IF
                { $<ttype>$ = c_begin_if_stmt (); }
235           '(' expr ')'
                { c_expand_start_cond (truthvalue_conversion ($4),
                                       compstmt_count,$<ttype>2);
                  $<itype>$ = stmt_count;
                  if_stmt_file = $<filename>-2;
                  if_stmt_line = $<lineno>-1; }
        ;

/* 이것은 stmt 의 하위 routine 이다.         이것은 두번 사용되는데,
   유효한 DO statement 들을 위해 한번, 그리고 끝 부분 test 를 해석
245   할 때 오류들을 잡기 위해 한번 사용된다.                    */
do_stmt_start:
        DO
                { stmt_count++;
                  compstmt_count++;
                  $<ttype>$
                    = add_stmt (build_stmt (DO_STMT, NULL_TREE,
                                            NULL_TREE));
                  /* Parse error 가 완전한 do-statement 를 해석하는
                     것을 방해하는 event 에서 codition 을 지금 설정한다.
                     그렇지 않을 경우, 우리는 RTL-generation 때 crash
                     할 것이다.       */
                  DO_COND ($<ttype>$) = error_mark_node; }
        c99_block_lineno_labeled_stmt WHILE
                { $$ = $<ttype>2;
                  RECHAIN_STMTS ($$, DO_BODY ($$)); }
        ;

for_init_stmt:
        xexpr ';'
                { add_stmt (build_stmt (EXPR_STMT, $1)); }
      | decl
                { check_for_loop_decls (); }
        ;

xexpr:
      /* empty */
                { $$ = NULL_TREE; }
      | expr
        ;
```

이제 아래부터 C 언어의 statement 표현식에 대해서 살펴보도록 하자. 우선적으로 IF 구문에 대해서 알아보도록 하자.

**(1)** if (1) { }

첫번째 예제로써, 가장 기본적인 if 표현식을 나타낸 것이다. 실제로 if 구문을 처리하게 되는 라벨은 select_or_iter_stmt 부터 시작한다고 할 수 있으며, IF token 을 만났을 때, c_begin_if_stmt () 함수를 통해서 IF_STMT node 를 만들고, (...) 구문을 처리한 후, c_expand_start_cond () 함수를 통해서, condition 을 확장하게 되며, 이 함수 부분에서 모호한 else 에 대한 평가가 이루

어지게 된다. 이제 모두 처리하였다면, c_finish_then () 함수가 호출되어, then-cluase 부분이 마치게 된다. 이제 뒤에 else 와 같은 구문이 오지 않기 때문에, 마지막으로 c_expand_end_cond () 함수를 통해서 구문 처리를 완료하게 된다.

```
compstmt_or_error -> compstmt
  compstmt -> compstmt_start compstmt_nostart
    compstmt_start -> '{'
    compstmt_nostart -> pushlevel maybe_label_decls
                        compstmt_contents_nonempty '}' poplevel
      pushlevel
      maybe_label_decls
      compstmt_contents_nonempty -> stmts_and_decls
        stmts_and_decls -> lineno_stmt_decl_or_labels_ending_stmt
          lineno_stmt_decl_or_labels_ending_stmt -> lineno_stmt
            lineno_stmt -> save_filename save_lineno stmt
              save_filename
              save_lineno
              stmt -> c99_block_start select_or_iter_stmt c99_block_end
                c99_block_start
                select_or_iter_stmt -> simple_if
                  simple_if -> if_prefix c99_block_lineno_labeled_stmt
                    if_prefix -> IF '(' expr ')'
                      expr -> nonnull_exprlist
                        nonnull_exprlist -> expr_no_commas
                          expr_no_commas -> cast_expr
                            cast_expr -> unary_expr
                              unary_expr -> primary
                                primary -> CONSTANT
                    c99_block_lineno_labeled_stmt -> c99_block_start
                                                     lineno_labeled_stmt
                                                     c99_block_end
                      c99_block_start
                      lineno_labeled_stmt -> lineno_stmt
                        lineno_stmt -> save_filename save_lineno stmt
                          save_filename
                          save_lineno
                          stmt -> compstmt
                            compstmt -> compstmt_start compstmt_nostart
                              compstmt_start -> '{'
                              compstmt_nostart -> '}'
                      c99_block_end
                c99_block_end
      poplevel
```

**(2)** if (1) { } else { }

두번째 예제는 예제는 위의 예제에 else 가 붙은 형태인데, 위의 구문 처리가 다른 부분이 있다면, select_or_iter_stmt 라벨에서 처리 부분일 것이다. else 가 시작되는 부분에, c_expand_start_else () 함수에 의해서 else 부분이 더 확장된다는 사실을 알리고, else 부분이 끝났을 때, c_finish_else () 함수를 호출한다. 완전히 if 구문이 끝났을 때는, 위의 예제와 마찬가지로 c_expand_end_cond () 함수를 호출하게 된다.

```
compstmt_or_error -> compstmt
  compstmt -> compstmt_start compstmt_nostart
    compstmt_start -> '{'
    compstmt_nostart -> pushlevel maybe_label_decls
                        compstmt_contents_nonempty '}' poplevel
```

```
pushlevel
maybe_label_decls
compstmt_contents_nonempty -> stmts_and_decls
  stmts_and_decls -> lineno_stmt_decl_or_labels_ending_stmt
    lineno_stmt_decl_or_labels_ending_stmt -> lineno_stmt
      lineno_stmt -> save_filename save_lineno stmt
        save_filename
        save_lineno
        stmt -> c99_block_start select_or_iter_stmt c99_block_end
          c99_block_start
          select_or_iter_stmt -> simple_if ELSE
                                 c99_block_lineno_labeled_stmt
            simple_if -> if_prefix c99_block_lineno_labeled_stmt
              if_prefix -> IF '(' expr ')'
                expr -> nonnull_exprlist
                  nonnull_exprlist -> expr_no_commas
                    expr_no_commas -> cast_expr
                      cast_expr -> unary_expr
                        unary_expr -> primary
                          primary -> CONSTANT
              c99_block_lineno_labeled_stmt -> c99_block_start
                                               lineno_labeled_stmt
                                               c99_block_end
                c99_block_start
                lineno_labeled_stmt -> lineno_stmt
                  lineno_stmt -> save_filename save_lineno stmt
                    save_filename
                    save_lineno
                    stmt -> compstmt
                      compstmt -> compstmt_start compstmt_nostart
                        compstmt_start -> '{'
                        compstmt_nostart -> '}'
                c99_block_end
            c99_block_lineno_labeled_stmt -> c99_block_start
                                             lineno_labeled_stmt
                                             c99_block_end
              c99_block_start
              lineno_labeled_stmt -> lineno_stmt
                lineno_stmt -> save_filename save_lineno stmt
                  save_filename
                  save_lineno
                  stmt -> compstmt
                    compstmt -> compstmt_start compstmt_nostart
                      compstmt_start -> '{'
                      compstmt_nostart -> '}'
              c99_block_end
          c99_block_end
poplevel
```

**(3)** if (1) { } else if (1) { }

세번째 예제는, 위의 예제에 if 구문을 하나 더 붙인 형태를 하고 있는데, Yacc 문법이 실제로 해석을 할 경우, 뒤에 나오는 if 구문은 앞의 else 내에 포함이 된 형태로 해석이 이루어진다고 할 수 있다. 즉,

```
if (1) {
```

```
    } else {
      if (1) {
      }
    }
```

와 같이 해석하는 것과 같게, 생각해서 처리하게 된다. 물론 {, } token 을 만났을 때 이루어지는 operation 들은 수행되지 않지만 말이다. 그래서 위의 두 예제와 실행되는 차례가 약간 더 복잡해 질 뿐 위에서 언급한 함수들이 고루 실행되어지게 된다. 자세한 operation 에 대한 설명은 다음 문서에서 이야기 할 것이다.

```
compstmt_or_error -> compstmt
  compstmt -> compstmt_start compstmt_nostart
    compstmt_start -> '{'
    compstmt_nostart -> pushlevel maybe_label_decls
                        compstmt_contents_nonempty '}' poplevel
      pushlevel
      maybe_label_decls
      compstmt_contents_nonempty -> stmts_and_decls
        stmts_and_decls -> lineno_stmt_decl_or_labels_ending_stmt
          lineno_stmt_decl_or_labels_ending_stmt -> lineno_stmt
            lineno_stmt -> save_filename save_lineno stmt
              save_filename
              save_lineno
              stmt -> c99_block_start select_or_iter_stmt c99_block_end
                c99_block_start
                select_or_iter_stmt -> simple_if ELSE
                                       c99_block_lineno_labeled_stmt
                  simple_if -> if_prefix c99_block_lineno_labeled_stmt
                    if_prefix -> IF '(' expr ')'
                      expr -> nonnull_exprlist
                        nonnull_exprlist -> expr_no_commas
                          expr_no_commas -> cast_expr
                            cast_expr -> unary_expr
                              unary_expr -> primary
                                primary -> CONSTANT
                    c99_block_lineno_labeled_stmt -> c99_block_start
                                                     lineno_labeled_stmt
                                                     c99_block_end
                      c99_block_start
                      lineno_labeled_stmt -> lineno_stmt
                        lineno_stmt -> save_filename save_lineno stmt
                          save_filename
                          save_lineno
                          stmt -> compstmt
                            compstmt -> compstmt_start compstmt_nostart
                              compstmt_start -> '{'
                              compstmt_nostart -> '}'
                      c99_block_end
                  c99_block_lineno_labeled_stmt -> c99_block_start
                                                   lineno_labeled_stmt
                                                   c99_block_end
                    c99_block_start
                    lineno_labeled_stmt -> lineno_stmt
                      lineno_stmt -> save_filename save_lineno stmt
                        save_filename
                        save_lineno
                        stmt -> c99_block_start select_or_iter_stmt
```

```
                                c99_block_end
                            select_or_iter_stmt -> simple_if
                              if_prefix c99_block_lineno_labeled_stmt
                                if_prefix -> IF '(' expr ')'
                                  expr -> nonnull_exprlist
                                    nonnull_exprlist -> expr_no_commas
                                      expr_no_commas -> cast_expr
                                        cast_expr -> unary_expr
                                          unary_expr -> primary
                                            primary -> CONSTANT
                                c99_block_lineno_labeled_stmt ->
                                    c99_block_start lineno_labeled_stmt
                                    c99_block_end
                                  c99_block_start
                                  lineno_labeled_stmt -> lineno_stmt
                                    lineno_stmt -> save_filename save_lineno stmt
                                      save_filename
                                      save_lineno
                                      stmt -> compstmt
                                        compstmt -> compstmt_start
                                                    compstmt_nostart
                                          compstmt_start -> '{'
                                          compstmt_nostart -> '}'

                                c99_block_end
                            c99_block_end
                        c99_block_end
                    c99_block_end
        poplevel
```

**(4)** for (; ;) { }

C 언어에서의 for 구문에 대해서 알아 보도록 하자. 중심이 되는 부분은 if 문과 마찬가지로 select_or_iter_stmt 라벨에서 모두 처리가 된다고 할 수 있겠고, FOR token 을 만났을 경우, FOR_STMT tree node 를 구성한 후 add_stmt () 로 statement-tree 에 더한다. for 구문의 경우, 조건이 ( ... ) 사이에 3 개가 존재하게 되는데, for_init_stmt 라벨과 xexpr 라벨이 그것이다. for_init_stmt 라벨의 경우, for 구문의 첫번째 조건에는 물론 표현식이 올 수 있지만, 변수에 대한 정의가 존재할 수 있기 때문에, 변수 처리도 가능하도록 만든 라벨이다. xexpr 은 expr 라벨을 포함하고 있지만, 표현식이 내부에 존재하지 않을 수도 있기 때문에 그에 대한 부분도 추가된 것을 말한다. 그외에 실제 { ... } 사이에 존재하는 statement 혹은 declaration 들은 c99_block_lineno_labeled_stmt 라벨에서 처리가 이루어지게 된다. select_or_iter_stmt 라벨을 보면 알겠지만, 생성되는 FOR_STMT tree node 의 operand 로써, FOR_COND 와 FOR_EXPR 가 설정이 되어진다는 것을 확인 할 수 있을 것이다. 물론 자세한 사항에 대해서는 다음 문서에서 언급할 것이다.

```
compstmt_or_error -> compstmt
  compstmt -> compstmt_start compstmt_nostart
    compstmt_start -> '{'
    compstmt_nostart -> pushlevel maybe_label_decls
                        compstmt_contents_nonempty '}' poplevel
      pushlevel
      maybe_label_decls
      compstmt_contents_nonempty -> stmts_and_decls
        stmts_and_decls -> lineno_stmt_decl_or_labels_ending_stmt
          lineno_stmt_decl_or_labels_ending_stmt -> lineno_stmt
            lineno_stmt -> save_filename save_lineno stmt
              save_filename
```

```
save_lineno
stmt -> c99_block_start select_or_iter_stmt c99_block_end
  c99_block_start
  select_or_iter_stmt -> FOR '(' for_init_stmt xexpr ';'
                         xexpr ')' c99_block_lineno_labeled_stmt
    for_init_stmt -> xexpr ';'
      xexpr
    xexpr
    xexpr
    c99_block_lineno_labeled_stmt -> c99_block_start
                                     lineno_labeled_stmt
                                     c99_block_end
      c99_block_start
      lineno_labeled_stmt -> lineno_stmt
        lineno_stmt -> save_filename save_lineno stmt
          save_filename
          save_lineno
           stmt -> compstmt
             compstmt -> compstmt_start compstmt_nostart
              compstmt_start -> '{'
              compstmt_nostart -> '}'
      c99_block_end
    c99_block_end
```

**(5)** do { } while (1);

위의 for 구문에서는 Yacc 문법의 처리 과정을 compstmt_or_error 라벨부터 보여주었지만, 아래부터는 select_or_iter_stmt 라벨을 기준으로 설명을 하도록 하겠다. 왜냐하면 compstmt_or_error 라벨부터 시작할 경우, 줄이 너무 길어지기 때문이다. 실제 do while 구문의 처리부분을 살펴보면, DO token 을 만났을 때, DO_STMT tree node 를 생성하고, DO_BODY 로써 c99_block_lineno_labeled_stmt 라벨에서 해석한 node 를 등록하게 된다. 그리고 되에 나오는 condition 에 대한 tree node 의 경우, DO_COND 에 등록이 되게 된다. 여기서 DO_BODY, DO_COND 는 Statement tree node 에 쉽게 접근을 하기 위한 accessor macro 이다.

```
select_or_iter_stmt -> do_stmt_start '(' expr ')' ';'
  expr -> nonnull_exprlist
    nonnull_exprlist -> expr_no_commas
      expr_no_commas -> cast_expr
        cast_expr -> unary_expr
          unary_expr -> primary
            primary -> CONSTANT
  do_stmt_start -> DO c99_block_lineno_labeled_stmt WHILE
    c99_block_lineno_labeled_stmt -> c99_block_start
                                     lineno_labeled_stmt c99_block_end
      c99_block_start
      lineno_labeled_stmt -> lineno_stmt
        lineno_stmt -> save_filename save_lineno stmt
          save_filename
          save_lineno
          stmt -> compstmt
            compstmt -> compstmt_start compstmt_nostart
              compstmt_start -> '{'
              compstmt_nostart -> '}'
      c99_block_end
```

**(6)** while (1) { }

위의 예제와 비슷하지만, 약간 실행 순서가 다른 while 구문에 대해서 알아보도록 하자. 이 while 구문의 처리 방법은 아래와 같다. WHILE token 을 만났을 때, c_begin_while_stmt () 함수를 호출하게 되고, condition 를 해석한 후, truthvalue_conversion () 함수를 거친 후, c_finish_while_stmt_cond () 함수를 호출함으로써, condition 을 마치게 되며, WHILE_BODY accessor 에 c99_block_lineno_labeled_stmt 라벨에서 해석한 tree node 를 등록함으로써, while 구문에 대한 해석이 마치게 된다.

```
select_or_iter_stmt -> WHILE '(' expr ')' c99_block_lineno_labeled_stmt
  expr -> nonnull_exprlist
    nonnull_exprlist -> expr_no_commas
      expr_no_commas -> cast_expr
        cast_expr -> unary_expr
          unary_expr -> primary
            primary -> CONSTANT
  c99_block_lineno_labeled_stmt -> c99_block_start lineno_labeled_stmt
                                   c99_block_end
    c99_block_start
    lineno_labeled_stmt -> lineno_stmt
      lineno_stmt -> save_filename save_lineno stmt
        save_filename
        save_lineno
        stmt -> compstmt
          compstmt -> compstmt_start compstmt_nostart
            compstmt_start -> '{'
            compstmt_nostart -> '}'
    c99_block_end
```

**(7)** switch (1) { case 'a': break; default: break; }

switch 구문에 대해서 알아보면, SWITCH token 을 만났을 때, c_start_case () 함수가 호출되어 초기화 되고, c99_block_lineno_labeled_stmt 라벨 수행이 끝난 후, c_finish_case () 함수가 호출되게 된다. c99_block_lineno_labeled_stmt 라벨에서 발생하는 operation 에 대해서 좀 더 알아보면, CASE token 을 만났을 때와, DEFAULT token 을 만났을 때, do_case () 함수가 호출되게 되며, BREAK token 을 만났을 때, build_break_stmt () 함수가 호출되어, statment tree 에 추가되게 된다.

```
select_or_iter_stmt -> SWITCH '(' expr ')' c99_block_lineno_labeled_stmt
  expr -> nonnull_exprlist
    nonnull_exprlist -> expr_no_commas
      expr_no_commas -> cast_expr
        cast_expr -> unary_expr
          unary_expr -> primary
            primary -> CONSTANT
  c99_block_lineno_labeled_stmt -> c99_block_start lineno_labeled_stmt
                                   c99_block_end
    c99_block_start
    lineno_labeled_stmt -> lineno_stmt
      lineno_stmt -> save_filename save_lineno stmt
        save_filename
        save_lineno
        stmt -> compstmt
          compstmt -> compstmt_start compstmt_nostart
            compstmt_start -> '{'
            compstmt_nostart -> pushlevel maybe_label_decls
                                compstmt_contents_nonempty '}' poplevel
              pushlevel
              maybe_label_decls
```

```
                compstmt_contents_nonempty -> stmts_and_decls
                  stmts_and_decls -> lineno_stmt_decl_or_labels_ending_stmt
                    lineno_stmt_decl_or_labels_ending_stmt ->
                        lineno_stmt_decl_or_labels_ending_label lineno_stmt
                      lineno_stmt_decl_or_labels_ending_label ->
                          lineno_stmt_decl_or_labels_ending_stmt lineno_label
                        lineno_stmt_decl_or_labels_ending_stmt ->
                            lineno_stmt_decl_or_labels_ending_label lineno_stmt
                          lineno_stmt_decl_or_labels_ending_label -> lineno_label
                            lineno_label -> save_filename save_lineno label
                              save_filename
                              save_lineno
                              label -> CASE expr_no_commas ':'
                                expr_no_commas -> cast_expr
                                  cast_expr -> unary_expr
                                    unary_expr -> primary
                                      primary -> CONSTANT
                            lineno_stmt -> save_filename save_lineno stmt
                              save_filename
                              save_lineno
                              stmt -> BREAK ';'
                          lineno_label -> save_filename save_lineno label
                            save_filename
                            save_lineno
                            label -> DEFAULT ':'
                        lineno_stmt -> save_filename save_lineno stmt
                          save_filename
                          save_lineno
                          stmt -> BREAK ';'
                poplevel
          c99_block_end
```

## 4.3 Declarator 표현식

Declarator 표현식의 경우, 위에서 언급한 "Data type 이 인식하는 문법" 에 나와 있는 yacc 문법을 모두 포함하는 표현식을 가지고 있으며, 이미 위에서 하위 라벨에 대해서는 언급하지 않아, 이것은 짧게 끝났다. 하지만 새로운 라벨 또한 존재하는데, *_nested_function 라벨이 그것이다.

```
decl:
      declspecs_ts setspecs initdecls ';'
            { POP_DECLSPEC_STACK; }
      | declspecs_nots setspecs notype_initdecls ';'
            { POP_DECLSPEC_STACK; }
      | declspecs_ts setspecs nested_function
            { POP_DECLSPEC_STACK; }
      | declspecs_nots setspecs notype_nested_function
            { POP_DECLSPEC_STACK; }
      | declspecs ';'
            { shadow_tag ($1); }
      | extension decl
            { RESTORE_WARN_FLAGS ($1); }
      ;

nested_function:
        declarator
            { if (pedantic)
```

```
                pedwarn ("ISO C forbids nested functions");

              push_function_context ();
              if (! start_function (current_declspecs, $1,
                                    all_prefix_attributes))
                {
                  pop_function_context ();
                  YYERROR1;
                }
            }
        old_style_parm_decls
            { store_parm_decls (); }
        save_filename save_lineno compstmt
            { tree decl = current_function_decl;
              DECL_SOURCE_FILE (decl) = $5;
              DECL_SOURCE_LINE (decl) = $6;
              finish_function (1, 1);
              pop_function_context ();
              add_decl_stmt (decl); }
      ;

notype_nested_function:
        notype_declarator
            { if (pedantic)
                pedwarn ("ISO C forbids nested functions");

              push_function_context ();
              if (! start_function (current_declspecs, $1,
                                    all_prefix_attributes))
                {
                  pop_function_context ();
                  YYERROR1;
                }
            }
        old_style_parm_decls
            { store_parm_decls (); }
        save_filename save_lineno compstmt
            { tree decl = current_function_decl;
              DECL_SOURCE_FILE (decl) = $5;
              DECL_SOURCE_LINE (decl) = $6;
              finish_function (1, 1);
              pop_function_context ();
              add_decl_stmt (decl); }
      ;

old_style_parm_decls:
      /* empty */
      | datadecls
      | datadecls ELLIPSIS
            /* ... 는 varargs function 를 가르키기 위해 여기 사용된다.              */
            { c_mark_varargs ();
              if (pedantic)
                pedwarn ("ISO C does not permit use of `varargs.h'"); }
      ;

/* 다음은 lineno_decl 와 decls, decl 와 닮은 꼴인데, 그들이 nested
   function 들을 허락하지 않는 것만 제외하고 말이다.                  그들은 old-style
```

```
  parm decls 을 위해 사용된다.        */
lineno_datadecl:
        save_filename save_lineno datadecl
             { }
      ;

datadecls:
      lineno_datadecl
      | errstmt
      | datadecls lineno_datadecl
      | lineno_datadecl errstmt
      ;

/* 우리는 여기에서 prefix attributes 를 허락하지 않는데, 그것은 reduce/reduce
  conflicts 를 일으키기 때문이다: 우리는 attribute suffix 를 가진 function decl
  를 해석하고 있는지, first old style parm 에 attribute prefix 를 가진
  function defn 를 해석하고 있는지 알 수 없다.              */
datadecl:
      declspecs_ts_nosa setspecs initdecls ';'
            { POP_DECLSPEC_STACK; }
      | declspecs_nots_nosa setspecs notype_initdecls ';'
            { POP_DECLSPEC_STACK; }
      | declspecs_ts_nosa ';'
            { shadow_tag_warned ($1, 1);
              pedwarn ("empty declaration"); }
      | declspecs_nots_nosa ';'
            { pedwarn ("empty declaration"); }
      ;
```

Declarator 표현식에 대한 예제는 앞서 이미 언급하였기 때문에, 여기에 다시 언급하지는 않겠다.

## 4.4 Label 표현식

Label 표현식은 기본적인 label 문을 포함하고 있으며, 또한 switch 구문에서 사용되는 case 문에 대한 statement 또한 label 로 인식한다.

```
/* 어떤 종류의 label.      Jump label 들과 case label 들을 또한 포함한다.
  ANSI C 는 statement 들 앞에만 label 들을 호용하지만 우리는 또한
  compound statement 끝에서도 그것을 허락한다.           */

label:  CASE expr_no_commas ':'
            { stmt_count++;
              $$ = do_case ($2, NULL_TREE); }
      | CASE expr_no_commas ELLIPSIS expr_no_commas ':'
            { stmt_count++;
              $$ = do_case ($2, $4); }
      | DEFAULT ':'
            { stmt_count++;
              $$ = do_case (NULL_TREE, NULL_TREE); }
      | identifier save_filename save_lineno ':' maybe_attribute
            { tree label = define_label ($2, $3, $1);
              stmt_count++;
              if (label)
                {
                  decl_attributes (&label, $5, 0);
                  $$ = add_stmt (build_stmt (LABEL_STMT, label));
```

```
            }
          else
            $$ = NULL_TREE;
        }
25      ;
```

그럼, 실제 label 이 어떻게 구성되게 되는지 아래에서 살펴보도록 하자.

int main ( ) { int i; lab: i++; goto lab; }

위의 예제는 라벨을 선언한 예제이다. 실제 라벨을 선언한 부분을 처리하게 되는 부분은 yacc 의 label 부분이며, 실제 라벨을 정의할 때, define_label () 함수를 통해서 정의하게 되며, 결과적으로 LABEL_STMT tree node 를 구성하게 된다. 또 실제로 goto 문을 사용하여 위에서 선언한 lab 이라는 라벨로 건너 뛰는 부분도 존재하게 되는 이에 대한 처리는 lookup_label () 함수를 통해서, 해당 라벨에 대한 decl 를 찾은 후, 결과적으로 GOTO_STMT tree node 를 구성하게 된다.

```
fndef -> declspecs_ts setspecs declarator old_style_parm_decls
         save_filename save_lineno compstmt_or_error
  declspecs_ts -> declspecs_nosc_ts_nosa_noea
    declspecs_nosc_ts_nosa_noea -> typespec_nonattr
      typespec_nonattr -> typespec_reserved_nonattr
        typespec_reserved_nonattr -> TYPESPEC
  setspecs
  declarator -> notype_declarator
    notype_declarator -> notype_declarator '(' parmlist_or_identifiers
      notype_declarator -> IDENTIFIER
      parmlist_or_identifiers -> maybe_attribute parmlist_or_identifiers_1
        maybe_attribute
        parmlist_or_identifiers_1 -> parmlist_1
          parmlist_1 -> parmlist_2 ')'
            parmlist_2
  old_style_parm_decls
  save_filename
  save_lineno
  compstmt_or_error -> compstmt
    compstmt -> compstmt_start compstmt_nostart
      compstmt_start -> '{'
      compstmt_nostart -> pushlevel maybe_label_decls
                          compstmt_contents_nonempty '}' poplevel
        pushlevel
        maybe_label_decls
        compstmt_contents_nonempty -> stmts_and_decls
          stmts_and_decls -> lineno_stmt_decl_or_labels_ending_stmt
            lineno_stmt_decl_or_labels_ending_stmt ->
                lineno_stmt_decl_or_labels_ending_stmt lineno_stmt
              lineno_stmt_decl_or_labels_ending_stmt ->
                  lineno_stmt_decl_or_labels_ending_label lineno_stmt
                lineno_stmt_decl_or_labels_ending_label ->
                    lineno_stmt_decl_or_labels_ending_decl lineno_label
                  lineno_stmt_decl_or_labels_ending_decl -> lineno_decl
                    lineno_decl -> save_filename save_lineno decl
                      save_filename
                      save_lineno
                      decl -> declspecs_ts setspecs initdecls ';'
                        declspecs_ts -> declspecs_nosc_ts_nosa_noea
                          declspecs_nosc_ts_nosa_noea -> typespec_nonattr
```

```
                        typespec_nonattr -> typespec_reserved_nonattr
                          typespec_reserved_nonattr -> TYPESPEC
                    setspecs
                    initdecls -> initdcl
                      initdcl -> declarator maybeasm maybe_attribute
                        declarator -> notype_declarator
                          notype_declarator -> IDENTIFIER
                        maybeasm
                        maybe_attribute
                  lineno_label -> save_filename save_lineno label
                    save_filename
                    save_lineno
                    label -> identifier save_filename save_lineno ':'
                             maybe_attribute
                      identifier -> IDENTIFIER
                      save_filename
                      save_lineno
                      maybe_attribute
                lineno_stmt -> save_filename save_lineno stmt
                  save_filename
                  save_lineno
                  stmt -> expr ';'
                    expr -> nonnull_exprlist
                      nonnull_exprlist -> expr_no_commas
                        expr_no_commas -> cast_expr
                          cast_expr -> unary_expr
                            unary_expr -> primary
                              primary -> primary PLUSPLUS
                                primary -> IDENTIFIER
          lineno_stmt -> save_filename save_lineno stmt
            save_filename
            save_lineno
            stmt -> GOTO identifier ';'
              identifier -> IDENTIFIER
    poplevel
```

## 제 5 절 Function

이 절에서는 GCC 에서 사용되는 함수 모형에 대해서 알아 보겠지만, 아래부터 나열되는 모든 하위 라벨들이 이미 위에서 한번쯤 언급한 것들로 구성되어 있다. 즉 위에서 언급한 라벨들의 조합으로 함수를 정의할 수 있음을 의미한다. 문법에 대해서는 위의 내용을 토대로 이해할 수 있을 것이기 때문에, 이제부터 중요한 것은 각각의 '상태'에 돌입했을 때, 어떠한 내부 작동이 이루어지느냐 일 것이며, 그에 대해서는 이제부터 하나씩 알아보도록 하겠다.

---

```
fndef:
        declspecs_ts setspecs declarator
             { if (! start_function (current_declspecs, $3,
                                     all_prefix_attributes))
                YYERROR1;
             }
        old_style_parm_decls
             { store_parm_decls (); }
        save_filename save_lineno compstmt_or_error
             { DECL_SOURCE_FILE (current_function_decl) = $7;
               DECL_SOURCE_LINE (current_function_decl) = $8;
```

```
                finish_function (0, 1);
                POP_DECLSPEC_STACK; }
        | declspecs_ts setspecs declarator error
                { POP_DECLSPEC_STACK; }
        | declspecs_nots setspecs notype_declarator
                { if (! start_function (current_declspecs, $3,
                                        all_prefix_attributes))
                    YYERROR1;
                }
          old_style_parm_decls
                { store_parm_decls (); }
          save_filename save_lineno compstmt_or_error
                { DECL_SOURCE_FILE (current_function_decl) = $7;
                  DECL_SOURCE_LINE (current_function_decl) = $8;
                  finish_function (0, 1);
                  POP_DECLSPEC_STACK; }
        | declspecs_nots setspecs notype_declarator error
                { POP_DECLSPEC_STACK; }
        | setspecs notype_declarator
                { if (! start_function (NULL_TREE, $2,
                                        all_prefix_attributes))
                    YYERROR1;
                }
          old_style_parm_decls
                { store_parm_decls (); }
          save_filename save_lineno compstmt_or_error
                { DECL_SOURCE_FILE (current_function_decl) = $6;
                  DECL_SOURCE_LINE (current_function_decl) = $7;
                  finish_function (0, 1);
                  POP_DECLSPEC_STACK; }
        | setspecs notype_declarator error
                { POP_DECLSPEC_STACK; }
        ;

/* 이것은 function definiton 의 몸체이다.
   다음 openbrace 를 무시하기 위해 syntax error 를 발생한다.              */
compstmt_or_error:
          compstmt
                {}
        | error compstmt
        ;
```

아래부터는 C 언어에서 함수를 선언할 때, 사용되는 기본적인 모형에 대해서 알아 보겠는데, 우선적으로는 함수의 prototype 을 선언할 때의 모형을 알아본 후, parameter 에 올 수 있는 모형을 하나 하나 살펴보면서 전개해 나가겠다.

**(1)** void abc (void);

첫번째 예제로써, 함수의 prototype 에 대해서 알아보도록 하자. 아래의 문법 해석이 어떻게 되는지 보면 알겠지만, 함수의 prototype 경우, fndef 라벨을 거치지 않고, 변수가 선언되는 것과 비슷하게 처리가 되어진다는 사실을 확인할 수 있다. 중요하게 되야 할 부분이 notype_declarator 라벨인데, 이 부분에서 함수를 위한 CALL_EXPR tree node 를 생성하게 된다. 실제 parameter 를 해석하게 되는 부분의 parmlist_or_identifiers 라벨도 중요하게 생각해야 할텐데, parameter 를 해석할 때 어떠한 operation 들이 이루어지는지 잘 봐야 할 것이다. 이에 대한 세부 작동은 다음 문서에서 언급이 될 것이다.

```
program -> extdefs
  extdefs -> extdef
    extdef -> datadef
      datadef -> declspecs_ts setspecs initdecls ';'
        declspecs_ts -> declspecs_nosc_ts_nosa_noea
          declspecs_nosc_ts_nosa_noea -> typespec_nonattr
            typespec_nonattr -> typespec_reserved_nonattr
              typespec_reserved_nonattr -> TYPESPEC
        setspecs
        initdecls -> initdcl
          initdcl -> declarator maybeasm maybe_attribute
            declarator -> notype_declarator
              notype_declarator -> notype_declarator '('
                                   parmlist_or_identifiers
                notype_declarator -> IDENTIFIER
                parmlist_or_identifiers -> maybe_attribute
                                           parmlist_or_identifiers_1
                  maybe_attribute
                  parmlist_or_identifiers_1 -> parmlist_1
                    parmlist_1 -> parmlist_2 ')'
                      parmlist_2 -> parms
                        parms -> firstparm
                          firstparm -> declspecs_ts_nosa setspecs_fp
                                       absdcl_maybe_attribute
                            declspecs_ts_nosa -> declspecs_nosc_ts_nosa_noea
                              declspecs_nosc_ts_nosa_noea -> typespec_nonattr
                                typespec_nonattr -> typespec_reserved_nonattr
                                  typespec_reserved_nonattr -> TYPESPEC
                            setspecs_fp -> setspecs
                              setspecs
                            absdcl_maybe_attribute
            maybeasm
            maybe_attribute
```

**(2)** int main ( ) { }

C 언어의 간단한 main 함수를 선언을 GCC 에서는 아래와 같이 처리를 한다.

```
program -> extdefs
  extdefs -> extdef
    extdef -> fndef
      fndef -> declspecs_ts setspecs declarator old_style_parm_decls
               save_filename save_lineno compstmt_or_error
        declspecs_ts -> declspecs_nosc_ts_nosa_noea
          declspecs_nosc_ts_nosa_noea -> typespec_nonattr
            typespec_nonattr -> typespec_reserved_nonattr
              typespec_reserved_nonattr -> TYPESPEC
        setspecs
        declarator -> notype_declarator
          notype_declarator -> notype_declarator '(' parmlist_or_identifiers
            notype_declarator -> IDENTIFIER
            parmlist_or_identifiers -> maybe_attribute parmlist_or_identifiers_1
              maybe_attribute
              parmlist_or_identifiers_1 -> parmlist_1
                parmlist_1 ->  parmlist_2 ')'
                  parmlist_2
        old_style_parm_decls
```

```
save_filename
save_lineno
compstmt_or_error -> compstmt
  compstmt -> compstmt_start compstmt_nostart
    compstmt_start -> '{'
    compstmt_nostart -> '}'
```

$prefix/gcc/c-parse.in 파일을 분석해 보면 알겠지만, 함수를 선언하는 fndef 라벨에서는 parameter 를 해석한 후, start_function () 함수를 통해서 함수의 시작을 알리고, 실제로 { ... } 가 끝나면, finish_function () 을 통해서 함수의 끝을 알리게 된다. 위의 진행도를 보면 알겠지만, 실제 함수의 parameter 는 declarator 라벨에서 처리가 된다는 사실을 확인할 수 있을 것이다. 하지만, C 언어 초기의 parameter style 의 경우, old_style_parm_decls 라벨에서 처리를 하게 된다. Old style 의 parameter 또한 처리가 끝날 경우, store_parm_decls () 함수를 통해서, 현재 처리한 parameter 를 현재 함수의 정의에 저장하게 된다.

**(3)** int main (int argc, char *argv[]) { }

세번째 예제로, 위의 예제와는 다르게, 실제로 parameter 를 선언한 형태를 가지고 있다. 앞에서 언급했듯이, Parameter 의 경우, declarator 라벨에서 처리가 된다. 각 parameter 들은 parm 라벨에서 처리가 된 후, push_parm_decl () 함수에 의해서 쌓이게 되며, get_parm_info () 함수에 의해서 구해진 parameter 들은, 나중에 CALL_EXPR 을 만들 때, 그에 대한 인자로 들어가게 된다.

```
program -> extdefs
  extdefs -> extdef
    extdef -> fndef
      fndef -> declspecs_ts setspecs declarator old_style_parm_decls
               save_filename save_lineno compstmt_or_error
        declspecs_ts -> declspecs_nosc_ts_nosa_noea
          declspecs_nosc_ts_nosa_noea -> typespec_nonattr
            typespec_nonattr -> typespec_reserved_nonattr
              typespec_reserved_nonattr -> TYPESPEC
        setspecs
        declarator -> notype_declarator
          notype_declarator -> notype_declarator '(' parmlist_or_identifiers
            notype_declarator -> IDENTIFIER
            parmlist_or_identifiers -> maybe_attribute parmlist_or_identifiers_1
              maybe_attribute
              parmlist_or_identifiers_1 -> parmlist_1
                parmlist_1 ->  parmlist_2 ')'
                  parmlist_2 -> parms
                    parms -> parms ',' parm
                      parms -> firstparm
                        firstparm -> declspecs_ts_nosa setspecs_fp
                                     notype_declarator maybe_attribute
                          declspecs_ts_nosa -> declspecs_nosc_ts_nosa_noea
                            declspecs_nosc_ts_nosa_noea -> typespec_nonattr
                              typespec_nonattr -> typespec_reserved_nonattr
                                typespec_reserved_nonattr -> TYPESPEC
                          setspecs_fp -> setspecs
                            setspecs
                          notype_declarator -> IDENTIFIER
                          maybe_attribute
                      parm -> declspecs_ts setspecs notype_declarator
                              maybe_attribute
                        declspecs_ts -> declspecs_nosc_ts_nosa_noea
```

```
                        declspecs_nosc_ts_nosa_noea -> typespec_nonattr
                          typespec_nonattr -> typespec_reserved_nonattr
                            typespec_reserved_nonattr -> TYPESPEC
                      setspecs
                      notype_declarator -> '*' maybe_type_quals_attrs
                                           notype_declarator
                        maybe_type_quals_attrs
                        notype_declarator -> notype_declarator array_declarator
                          notype_declarator -> IDENTIFIER
                          array_declarator -> '[' ']'
                      maybe_attribute
          old_style_parm_decls
          save_filename
          save_lineno
          compstmt_or_error -> compstmt
            compstmt -> compstmt_start compstmt_nostart
              compstmt_start -> '{'
              compstmt_nostart -> '}'
```

**(4)** int abc (int a, ...) { }

이 예제는 위와 바른 점이 있다면, ELLIPSIS 가 존재하는 것일 것이다. GCC 에서 이러한 ELLIPSIS 를 만날 때, 다른 인자와 다른 점은 get_parm_info () 함수를 호출할 때, ELLIPSIS 나 나올 경우, 인자로 0 을 주며, 아닐 경우, 1 을 준다는 점이다.

```
program -> extdefs
  extdefs -> extdef
    extdef -> fndef
      fndef -> declspecs_ts setspecs declarator old_style_parm_decls
               save_filename save_lineno compstmt_or_error
        declspecs_ts -> declspecs_nosc_ts_nosa_noea
          declspecs_nosc_ts_nosa_noea -> typespec_nonattr
            typespec_nonattr -> typespec_reserved_nonattr
              typespec_reserved_nonattr -> TYPESPEC
        setspecs
        declarator -> notype_declarator
          notype_declarator -> notype_declarator '(' parmlist_or_identifiers
            notype_declarator -> IDENTIFIER
            parmlist_or_identifiers -> maybe_attribute parmlist_or_identifiers_1
              maybe_attribute
              parmlist_or_identifiers_1 -> parmlist_1
                parmlist_1 ->  parmlist_2 ')'
                  parmlist_2 -> parms ',' ELLIPSIS
                    parms -> firstparm
                      firstparm -> declspecs_ts_nosa setspecs_fp
                                   notype_declarator maybe_attribute
                        declspecs_ts_nosa -> declspecs_nosc_ts_nosa_noea
                          declspecs_nosc_ts_nosa_noea -> typespec_nonattr
                            typespec_nonattr -> typespec_reserved_nonattr
                              typespec_reserved_nonattr -> TYPESPEC
                        setspecs_fp -> setspecs
                            setspecs
                        notype_declarator -> IDENTIFIER
                        maybe_attribute
        old_style_parm_decls
        save_filename
        save_lineno
```

```
          compstmt_or_error -> compstmt
            compstmt -> compstmt_start compstmt_nostart
              compstmt_start -> '{'
              compstmt_nostart -> '}'
```

**(5)** int abc (a) int a; { }

이 예제의 경우, Old style parameter 가 올 경우에 대한 것으로써, 다른 예제와 다른 점은 old_style_parm_decls 라벨에서의 처리일 것이다. 대부분의 다른 예제는 old_style_parm_decls 라벨에서 그냥 넘어가지만, 실제로 이 예제에서는 int a; 와 같은 변수 선언이 존재하기 때문에, 실제 ( ... ) 로 둘러쌓여있는 변수와, ')' 뒤에 오는 변수의 선언이 어떻게 이어지는지에 대해 살펴볼 필요성이 있을 것이다. 세부적인 것에 대해서는 다음 문서에서 언급할 예정이다.

```
program -> extdefs
  extdefs -> extdef
    extdef -> fndef
      fndef -> declspecs_ts setspecs declarator old_style_parm_decls
               save_filename save_lineno compstmt_or_error
        declspecs_ts -> declspecs_nosc_ts_nosa_noea
          declspecs_nosc_ts_nosa_noea -> typespec_nonattr
            typespec_nonattr -> typespec_reserved_nonattr
              typespec_reserved_nonattr -> TYPESPEC
        setspecs
        declarator -> notype_declarator
          notype_declarator -> notype_declarator '(' parmlist_or_identifiers
            notype_declarator -> IDENTIFIER
            parmlist_or_identifiers -> maybe_attribute
                                       parmlist_or_identifiers_1
              maybe_attribute
              parmlist_or_identifiers_1 -> identifiers ')'
                identifiers -> IDENTIFIER
        old_style_parm_decls -> datadecls
          datadecls -> lineno_datadecl
            lineno_datadecl -> save_filename save_lineno datadecl
              save_filename
              save_lineno
              datadecl -> declspecs_ts_nosa setspecs initdecls ';'
                declspecs_ts_nosa -> declspecs_nosc_ts_nosa_noea
                  declspecs_nosc_ts_nosa_noea -> typespec_nonattr
                    typespec_nonattr -> typespec_reserved_nonattr
                      typespec_reserved_nonattr -> TYPESPEC
                setspecs
                initdecls -> initdcl
                  initdcl -> declarator maybeasm maybe_attribute
                    declarator -> notype_declarator
                      notype_declarator -> IDENTIFIER
                    maybeasm
                    maybe_attribute
        save_filename
        save_lineno
        compstmt_or_error -> compstmt
          compstmt -> compstmt_start compstmt_nostart
            compstmt_start -> '{'
            compstmt_nostart -> '}'
```

위의 예제를 통해서 알 수 있겠지만, GCC 함수를 선언하는 것은 parameter 를 제외하고는 start_function 혹은 finish_function 으로 처리된다는 사실을 알 수 있겠고, { 혹은 } 같은 대괄호의 경우 COMPOUND_STMT 가 생성된다는 것을 알 수 있을 것이다.

## 제 6 절 22 주 문서를 마치며

이번 주 문서에서는, GCC 에서 사용되는 C 언어를 위한 yacc 문법에 대해서 알아보았으며, 각 정의나, 선언, 표현식 등등을 GCC 에서 처리할 때, 어떠한 함수들이 호출되는지, 어떠한 순서대로 처리되게 되는지에 대해서 간략하게 알아보았다. 다음 문서에서는 실제가 각각의 표현식을 정의할 때, GCC 에서 어떻게 이러한 표현식을 처리하는지 좀 더 자세하게 알아보도록 하겠다.