# GCC LALR Syntax (1)
# C-Lex

정원교

2004년 3월 24일

## 목 차

**제 1 절 21 주 문서를 시작하며** **2**

**제 2 절 Token** **2**
- 2.1 Yacc 를 위한 token 들 . . . 2
- 2.2 C 혹은 C++ 를 위한 토큰 . . . 4

**제 3 절 예약어** **6**
- 3.1 등록에 앞서 . . . 8
- 3.2 예약어의 등록 . . . 13

**제 4 절 yylex () 함수** **13**
- 4.1 c_lex () 함수 . . . 14
  - 4.1.1 lex_number . . . 15
  - 4.1.2 lex_charconst . . . 17
  - 4.1.3 lex_string . . . 17
- 4.2 cpp_get_token () 함수 . . . 18
- 4.3 Number 와 String, Identifier 의 처리 . . . 24

**제 5 절 Token 의 관리** **24**

**제 6 절 Macro 의 처리** **25**
- 6.1 Directive 의 구성 . . . 25
- 6.2 Directive 의 등록 . . . 27
- 6.3 처리한 directive 저장소들 . . . 28
- 6.4 Directive 의 처리 . . . 29
- 6.5 Directive handler . . . 30
- 6.6 처리한 directive 의 적용 . . . 31

# 제 1 절 21 주 문서를 시작하며

이번 주부터는 GCC 의 Yacc 문법과 결합되어 있는 Lex 에 대해서 알아 보도록 합니다. 기본적인 Lex 및 Yacc 의 사용법에 대해서는 이미 알고 있다는 가정하에, GCC 에서는 어떻게 Lex 를 구성하고 있는지를 살펴보도록 하겠습니다.

# 제 2 절 Token

GCC 에는 두 종류의 token 을 사용하는데, 하나는 yacc 를 위한 token 이고 나머지 하나는 C 혹은 C++ 를 위한 token 이다. C 언어 입장에서 보았을 때는 C 용 token 을 통해 표현되어지며, 실제 DFA 에 기반하여 status 변화를 일으키기 위해서는 yacc token 을 사용해야 한다. 즉 C 용 token 을 Lexer 가 인식해서 최종적으로 yacc 에서 넘겨 줄 때는 yacc 용 token 으로 변환시켜 주게 된다.

아래의 하위 섹션에서 어떻게 각 token 들이 구성되어 있는지 살펴보도록 하자.

## 2.1 Yacc 를 위한 token 들

Yacc 입장에 보았을 때 사용하는 token 들을 아래에 나열하였다.

- IDENTIFIER

  예약어가 아니고 현재 block 내 typedef 들로 정의된 것이 아닌 모든 identifier 들.

- TYPENAME

  현재 block 내 typedef 로 정의된 모든 identifier 들. 몇몇 constext 내에서는, 그들이 IDENTIFIER 와 같이 취급 받을 수 있는데, 하지만 그들은 또한 declaration 들내 typespec 들로써 제공할 수 있다.

- SCSPEC

  Storage class 를 지정하는 예약어. yylval 는 위의 것을 가르키는 IDENTIFIER_NODE 를 포함하고 있다.

- TYPESPEC

  Type 을 지정하는 예약어 yylval 는 위의 것을 가르키는 IDENTIFIER_NODE 를 포함하고 있다.

- TYPE_QUAL

  ”const” 혹은 ”volatile”, ”restrict” 과 같은 type 을 한정하는 예약어 yylval 는 위의 것을 가르키는 IDENTIFIER_NODE 를 포함하고 있다.

- CONSTANT

  문자 혹은 숫자 상수들. yylval 는 상수를 위한 node 이다.

- STRING

  Raw form 형식인 문자열 상수들. yylval 는 STRING_CST node 이다.

- ELLIPSIS

  변수 arglist 들을 가지는 함수에 사용되는 ”...”

- SIZEOF
  ENUM
  STRUCT
  UNION
  IF
  ELSE
  WHILE
  DO
  FOR
  SWITCH
  CASE
  DEFAULT
  BREAK
  CONTINUE
  RETURN
  GOTO
  ASM_KEYWORD
  TYPEOF
  ALIGNOF
  ATTRIBUTE
  EXTENSION
  LABEL
  REALPART
  IMAGPART
  VA_ARG
  CHOOSE_EXPR
  TYPES_COMPATIBLE_P
  PTR_VALUE
  PTR_BASE
  PTR_EXTENT

  예약어들

- STRING_FUNC_NAME
  VAR_FUNC_NAME

  Function 이름은 string const 혹은 var decl 일 수 있다.

- INTERFACE
  IMPLEMENTATION
  END
  SELECTOR
  DEFS
  ENCODE
  CLASSNAME
  PUBLIC
  PRIVATE
  PROTECTED
  PROTOCOL
  OBJECTNAME
  CLASS
  ALIAS

  Objective-C 키워드들. 이것은 C 와 Objective C 에 포함되며, token code 들은 둘다 같다.

- ASSIGN
- OROR
- ANDAND
- EQCOMPARE
- ARITHCOMPARE
- LSHIFT
  RSHIFT
- UNARY
  PLUSPLUS
  MINUSMINUS
- HYPERUNARY
- POINTSAT
- =
  !
  +
  −
  *
  /
  %
  &
  |
  ^
  ~
  ?
  (
  [
  ]
  {
  }
  .
  :
  ,
  )
  ;

  C 언어에서 사용되는 문자 토큰들

## 2.2 C 혹은 C++ 를 위한 토큰

GCC 에서 언어 C 를 위해 사용되는 token 이 존재한다. C 언어의 각 표현식에서 사용되는 표현들은 아래에 나와 있는 token 들로 표현되어진다고 할 수 있다.

C 혹은 C++ 용 token 은 열거자 cpp_ttype 내에 모두 나타나 있으며, C 언어의 표현은 이를 벗어나지 않는다.

---

```
#define CPP_LAST_EQ CPP_MAX
#define CPP_FIRST_DIGRAPH CPP_HASH
#define CPP_LAST_PUNCTUATOR CPP_DOT_STAR
```

```c
#define TTYPE_TABLE                                   \
  OP(CPP_EQ = 0,      "=")                            \
  OP(CPP_NOT,         "!")                            \
  OP(CPP_GREATER,     ">")   /* 비교 */               \
  OP(CPP_LESS,        "<")                            \
  OP(CPP_PLUS,        "+")   /* 연산 */               \
  OP(CPP_MINUS,       "-")                            \
  OP(CPP_MULT,        "*")                            \
  OP(CPP_DIV,         "/")                            \
  OP(CPP_MOD,         "%")                            \
  OP(CPP_AND,         "&")   /* bit 관련 */           \
  OP(CPP_OR,          "|")                            \
  OP(CPP_XOR,         "^")                            \
  OP(CPP_RSHIFT,      ">>")                           \
  OP(CPP_LSHIFT,      "<<")                           \
  OP(CPP_MIN,         "<?")  /* 확장 */               \
  OP(CPP_MAX,         ">?")                           \
\
  OP(CPP_COMPL,       "~")                            \
  OP(CPP_AND_AND,     "&&")  /* 논리 */               \
  OP(CPP_OR_OR,       "||")                           \
  OP(CPP_QUERY,       "?")                            \
  OP(CPP_COLON,       ":")                            \
  OP(CPP_COMMA,       ",")   /* 그룹화 */             \
  OP(CPP_OPEN_PAREN,  "(")                            \
  OP(CPP_CLOSE_PAREN, ")")                            \
  OP(CPP_EQ_EQ,       "==")  /* 비교 */               \
  OP(CPP_NOT_EQ,      "!=")                           \
  OP(CPP_GREATER_EQ,  ">=")                           \
  OP(CPP_LESS_EQ,     "<=")                           \
\
  OP(CPP_PLUS_EQ,     "+=")  /* 연산 */               \
  OP(CPP_MINUS_EQ,    "-=")                           \
  OP(CPP_MULT_EQ,     "*=")                           \
  OP(CPP_DIV_EQ,      "/=")                           \
  OP(CPP_MOD_EQ,      "%=")                           \
  OP(CPP_AND_EQ,      "&=")  /* bit 관련 */           \
  OP(CPP_OR_EQ,       "|=")                           \
  OP(CPP_XOR_EQ,      "^=")                           \
  OP(CPP_RSHIFT_EQ,   ">>=")                          \
  OP(CPP_LSHIFT_EQ,   "<<=")                          \
  OP(CPP_MIN_EQ,      "<?=") /* 확장 */               \
  OP(CPP_MAX_EQ,      ">?=")                          \
  /* CPP_FIRST_DIGRAPH 로 시작하는 digraph 들.   */   \
  OP(CPP_HASH,        "#")   /* digraphs */           \
  OP(CPP_PASTE,       "##")                           \
  OP(CPP_OPEN_SQUARE, "[")                            \
  OP(CPP_CLOSE_SQUARE, "]")                           \
  OP(CPP_OPEN_BRACE,  "{")                            \
  OP(CPP_CLOSE_BRACE, "}")                            \
  /* 구두점에 대한 remainder.    순서는 상관없음.    */ \
  OP(CPP_SEMICOLON,   ";")   /* 구조체 */             \
  OP(CPP_ELLIPSIS,    "...")                          \
  OP(CPP_PLUS_PLUS,   "++")  /* 증가 */               \
  OP(CPP_MINUS_MINUS, "--")                           \
  OP(CPP_DEREF,       "->")  /* 접근자 */             \
```

```
  OP(CPP_DOT,        ".")                  \
  OP(CPP_SCOPE,      "::")                 \
  OP(CPP_DEREF_STAR, "->*")                \
  OP(CPP_DOT_STAR,   ".*")                 \
65  OP(CPP_ATSIGN,     "@") /* Object C 에서 사용 */ \
\
  TK(CPP_NAME,       SPELL_IDENT) /* 단어 */          \
  TK(CPP_NUMBER,     SPELL_NUMBER) /* 34_be+ta */     \
\
70  TK(CPP_CHAR,       SPELL_STRING) /* 'char' */       \
  TK(CPP_WCHAR,      SPELL_STRING) /* L'char' */      \
  TK(CPP_OTHER,      SPELL_CHAR)  /* stray 구두점 */        \
\
  TK(CPP_STRING,     SPELL_STRING) /* ``string'' */     \
75  TK(CPP_WSTRING,    SPELL_STRING) /* L``string'' */    \
  TK(CPP_HEADER_NAME, SPELL_STRING) /* #include 에서의 <stdio.h> */ \
\
  TK(CPP_COMMENT,    SPELL_NUMBER) /* Only if output comments. */ \
                                  /* SPELL_NUMBER happens to DTRT. */ \
80  TK(CPP_MACRO_ARG, SPELL_NONE)  /* Macro 인자.  */        \
  TK(CPP_PADDING,   SPELL_NONE)  /* cpp0 용 whitespace. */   \
  TK(CPP_EOF,       SPELL_NONE)  /* 파일 혹은 줄의 끝.        */

#define OP(e, s) e,
85  #define TK(e, s) e,
enum cpp_ttype
{
  TTYPE_TABLE
  N_TTYPES
90  };
#undef OP
#undef TK
```

# 제 3 절 예약어

예약어는 "Reserved Words" 들을 가르킨다는 것을 잘 알고 있을 것이다. GCC 에서는 각 언어 마다의 예약어들을 가지고 있다. 이 예약어에 대해서 살펴보고 넘어 가도록 하자.

예약어 목록은 $prefix/gcc/c-parse.in 파일에 선언되어 있는 전역 변수 struct resword reswords[] 에 존재한다. 이 예약어를 구성하는 구조체에 대해 잠시 설명하고 넘어 가면 아래와 같다.

```
struct resword
{
  const char *word;
  ENUM_BITFIELD(rid) rid : 16;
  unsigned int disable   : 16;
};
```

구성요소인 word 는 예약어의 이름, rid 는 RID 의 분류값, disable 은 이 예약어를 실제로 사용할 것인지를 나타낸다.

잠시 위 disable 구성요소에 대해 살펴본 후 넘어가면,

```
#define D_TRAD  0x01    /* Traditional C 에서 설정 안됨 */
#define D_C89   0x02    /* C89 에서 설정 안됨 */
```

```
#define D_EXT   0x04    /* GCC extension */
#define D_EXT89 0x08    /* C99 와 변환된 GCC extension */
#define D_OBJC  0x10    /* 오직 Objective C */
```

위와 같은 mask 값들을 가지고 되어 있다. 실제 예약어는

```
(reswords[i].disable & mask)
```

값이 참일 경우 예약어로써 등록안되게 구성되어 있는데, 위의 define 으로 선언된 것은 mask 형태를 뛰기 때문에, &, | 와 같은 AND, OR 연산을 통해 여러개를 동시에 지정할 수 있다.

C 에서는 어떤 예약어가 존재할 것인가. 그것은 아래에 있다.

```
static const struct resword reswords[] =
{
  { "_Bool",        RID_BOOL,     0 },
  { "_Complex",     RID_COMPLEX, 0 },
  { "__FUNCTION__", RID_FUNCTION_NAME, 0 },
  { "__PRETTY_FUNCTION__", RID_PRETTY_FUNCTION_NAME, 0 },
  { "__alignof",    RID_ALIGNOF, 0 },
  { "__alignof__",  RID_ALIGNOF, 0 },
  { "__asm",        RID_ASM,     0 },
  { "__asm__",      RID_ASM,     0 },
  { "__attribute",  RID_ATTRIBUTE, 0 },
  { "__attribute__", RID_ATTRIBUTE, 0 },
  { "__bounded",    RID_BOUNDED, 0 },
  { "__bounded__",  RID_BOUNDED, 0 },
  { "__builtin_choose_expr", RID_CHOOSE_EXPR, 0 },
  { "__builtin_types_compatible_p", RID_TYPES_COMPATIBLE_P, 0 },
  { "__builtin_va_arg", RID_VA_ARG, 0 },
  { "__complex",    RID_COMPLEX, 0 },
  { "__complex__",  RID_COMPLEX, 0 },
  { "__const",      RID_CONST,   0 },
  { "__const__",    RID_CONST,   0 },
  { "__extension__", RID_EXTENSION, 0 },
  { "__func__",     RID_C99_FUNCTION_NAME, 0 },
  { "__imag",       RID_IMAGPART, 0 },
  { "__imag__",     RID_IMAGPART, 0 },
  { "__inline",     RID_INLINE,  0 },
  { "__inline__",   RID_INLINE,  0 },
  { "__label__",    RID_LABEL,   0 },
  { "__ptrbase",    RID_PTRBASE, 0 },
  { "__ptrbase__",  RID_PTRBASE, 0 },
  { "__ptrextent",  RID_PTREXTENT, 0 },
  { "__ptrextent__", RID_PTREXTENT, 0 },
  { "__ptrvalue",   RID_PTRVALUE, 0 },
  { "__ptrvalue__", RID_PTRVALUE, 0 },
  { "__real",       RID_REALPART, 0 },
  { "__real__",     RID_REALPART, 0 },
  { "__restrict",   RID_RESTRICT, 0 },
  { "__restrict__", RID_RESTRICT, 0 },
  { "__signed",     RID_SIGNED,  0 },
  { "__signed__",   RID_SIGNED,  0 },
  { "__typeof",     RID_TYPEOF,  0 },
  { "__typeof__",   RID_TYPEOF,  0 },
  { "__unbounded",  RID_UNBOUNDED, 0 },
  { "__unbounded__", RID_UNBOUNDED, 0 },
  { "__volatile",   RID_VOLATILE, 0 },
```

```
    { "__volatile__", RID_VOLATILE, 0 },
    { "asm",          RID_ASM,      D_EXT },
    { "auto",         RID_AUTO,     0 },
    { "break",        RID_BREAK,    0 },
50  { "case",         RID_CASE,     0 },
    { "char",         RID_CHAR,     0 },
    { "const",        RID_CONST,    D_TRAD },
    { "continue",     RID_CONTINUE, 0 },
    { "default",      RID_DEFAULT,  0 },
55  { "do",           RID_DO,       0 },
    { "double",       RID_DOUBLE,   0 },
    { "else",         RID_ELSE,     0 },
    { "enum",         RID_ENUM,     0 },
    { "extern",       RID_EXTERN,   0 },
60  { "float",        RID_FLOAT,    0 },
    { "for",          RID_FOR,      0 },
    { "goto",         RID_GOTO,     0 },
    { "if",           RID_IF,       0 },
    { "inline",       RID_INLINE,   D_TRAD|D_EXT89 },
65  { "int",          RID_INT,      0 },
    { "long",         RID_LONG,     0 },
    { "register",     RID_REGISTER, 0 },
    { "restrict",     RID_RESTRICT, D_TRAD|D_C89 },
    { "return",       RID_RETURN,   0 },
70  { "short",        RID_SHORT,    0 },
    { "signed",       RID_SIGNED,   D_TRAD },
    { "sizeof",       RID_SIZEOF,   0 },
    { "static",       RID_STATIC,   0 },
    { "struct",       RID_STRUCT,   0 },
75  { "switch",       RID_SWITCH,   0 },
    { "typedef",      RID_TYPEDEF,  0 },
    { "typeof",       RID_TYPEOF,   D_TRAD|D_EXT },
    { "union",        RID_UNION,    0 },
    { "unsigned",     RID_UNSIGNED, 0 },
80  { "void",         RID_VOID,     0 },
    { "volatile",     RID_VOLATILE, D_TRAD },
    { "while",        RID_WHILE,    0 },
};
```

낯익은 단어들이 들어가 있다는 사실을 알 수 있을 텐데, 위의 문자열들이 현재 GCC 의 C 언어 파트에서 예약어인 단어들이다.

## 3.1 등록에 앞서

위 정의된 전역 변수 reswords 를 보게 되면, RID_... 로 시작하는 많은 열거자 를 볼 수 있는데, 이 의미를 살펴본 후 넘어가도록 하자.

이 열거자들은 $prefix/gcc/c-common.h 에 선언되어 있는 것으로 예약된 식별자들이다. 이것은 C 와, C++, Objective C 에서 사용되는 모든 key 값들의 조합을 나타내고 있다.

이것의 사용 목적은 GCC 에서 사용되는 예약어들을 위한 목록이라고 할 수 있다. 위에서 언급한 cpp_token 은 예약어에 대해서도 CPP_NAME 으로 판단되기 때문에 예약어와 실제 identifier 를 구분해 줄 수 있는 구분 조건이 필요하다.

```
enum rid
{
```

```
/* Modifiers: */
/* C : 경험적 빈도수에 따른 순서.          */
RID_STATIC = 0,
RID_UNSIGNED, RID_LONG, RID_CONST, RID_EXTERN,
RID_REGISTER, RID_TYPEDEF, RID_SHORT, RID_INLINE,
RID_VOLATILE, RID_SIGNED, RID_AUTO, RID_RESTRICT,

/* C 확장용 */
RID_BOUNDED, RID_UNBOUNDED, RID_COMPLEX,

/* C++ */
RID_FRIEND, RID_VIRTUAL, RID_EXPLICIT, RID_EXPORT, RID_MUTABLE,

/* ObjC */
RID_IN, RID_OUT, RID_INOUT, RID_BYCOPY, RID_BYREF, RID_ONEWAY,

/* C */
RID_INT,   RID_CHAR, RID_FLOAT, RID_DOUBLE, RID_VOID,
RID_ENUM,  RID_STRUCT, RID_UNION, RID_IF, RID_ELSE,
RID_WHILE, RID_DO,   RID_FOR,    RID_SWITCH, RID_CASE,
RID_DEFAULT, RID_BREAK, RID_CONTINUE, RID_RETURN, RID_GOTO,
RID_SIZEOF,

/* C 확장판들 */
RID_ASM,     RID_TYPEOF, RID_ALIGNOF, RID_ATTRIBUTE, RID_VA_ARG,
RID_EXTENSION, RID_IMAGPART, RID_REALPART, RID_LABEL, RID_PTRBASE,
RID_PTREXTENT, RID_PTRVALUE, RID_CHOOSE_EXPR, RID_TYPES_COMPATIBLE_P,

/* 너무 많은 문자열토수뻘s 함수의 이름을 얻는 방법

   Too many ways of getting the name of a function as a string */
RID_FUNCTION_NAME, RID_PRETTY_FUNCTION_NAME, RID_C99_FUNCTION_NAME,

/* C++ */
RID_BOOL,   RID_WCHAR, RID_CLASS,
RID_PUBLIC, RID_PRIVATE, RID_PROTECTED,
RID_TEMPLATE, RID_NULL, RID_CATCH,
RID_DELETE, RID_FALSE, RID_NAMESPACE,
RID_NEW,    RID_OPERATOR, RID_THIS,
RID_THROW, RID_TRUE,   RID_TRY,
RID_TYPENAME, RID_TYPEID, RID_USING,

/* Cast 들

   casts */
RID_CONSTCAST, RID_DYNCAST, RID_REINTCAST, RID_STATCAST,

/* 동일한 철자들

   alternate spellings */
RID_AND, RID_AND_EQ, RID_NOT, RID_NOT_EQ,
RID_OR, RID_OR_EQ, RID_XOR, RID_XOR_EQ,
RID_BITAND, RID_BITOR, RID_COMPL,

/* Objective C */
RID_ID,        RID_AT_ENCODE, RID_AT_END,
RID_AT_CLASS, RID_AT_ALIAS,  RID_AT_DEFS,
```

```
60   RID_AT_PRIVATE, RID_AT_PROTECTED, RID_AT_PUBLIC,
     RID_AT_PROTOCOL, RID_AT_SELECTOR, RID_AT_INTERFACE,
     RID_AT_IMPLEMENTATION,

     RID_MAX,
65
     RID_FIRST_MODIFIER = RID_STATIC,
     RID_LAST_MODIFIER = RID_ONEWAY,

     RID_FIRST_AT = RID_AT_ENCODE,
70   RID_LAST_AT = RID_AT_IMPLEMENTATION,
     RID_FIRST_PQ = RID_IN,
     RID_LAST_PQ = RID_ONEWAY
   };
```

위의 리스트는 그 분류 RID 의 전체적인 윤곽이다. 그럼 현재 GCC 의 C 언어는 이 RID 를 yacc token 번호로 어떻게 변환하고 있는지에 대해 알아봐야 할 것이다.

그 해당 변환 table 은 $prefix/gcc/c-parse.in 에 선언되어 있는 전역 변수

`static const short rid_to_yy[RID_MAX]`

에 열거되어 있으며, 다음과 같이 매치됨을 알기 바란다.

```
   static const short rid_to_yy[RID_MAX] =
   {
     /* RID_STATIC */   SCSPEC,
     /* RID_UNSIGNED */ TYPESPEC,
 5   /* RID_LONG */     TYPESPEC,
     /* RID_CONST */    TYPE_QUAL,
     /* RID_EXTERN */   SCSPEC,
     /* RID_REGISTER */ SCSPEC,
     /* RID_TYPEDEF */  SCSPEC,
10   /* RID_SHORT */    TYPESPEC,
     /* RID_INLINE */   SCSPEC,
     /* RID_VOLATILE */ TYPE_QUAL,
     /* RID_SIGNED */   TYPESPEC,
     /* RID_AUTO */     SCSPEC,
15   /* RID_RESTRICT */ TYPE_QUAL,

     /* C 확장들
        C extensions */
     /* RID_BOUNDED */ TYPE_QUAL,
20   /* RID_UNBOUNDED */ TYPE_QUAL,
     /* RID_COMPLEX */ TYPESPEC,

     /* C++ */
     /* RID_FRIEND */   0,
25   /* RID_VIRTUAL */  0,
     /* RID_EXPLICIT */ 0,
     /* RID_EXPORT */   0,
     /* RID_MUTABLE */  0,

30   /* ObjC */
     /* RID_IN */       TYPE_QUAL,
     /* RID_OUT */      TYPE_QUAL,
     /* RID_INOUT */    TYPE_QUAL,
```

```
    /* RID_BYCOPY */  TYPE_QUAL,
35  /* RID_BYREF */   TYPE_QUAL,
    /* RID_ONEWAY */  TYPE_QUAL,

    /* C */
    /* RID_INT */     TYPESPEC,
40  /* RID_CHAR */    TYPESPEC,
    /* RID_FLOAT */   TYPESPEC,
    /* RID_DOUBLE */  TYPESPEC,
    /* RID_VOID */    TYPESPEC,
    /* RID_ENUM */    ENUM,
45  /* RID_STRUCT */  STRUCT,
    /* RID_UNION */   UNION,
    /* RID_IF */      IF,
    /* RID_ELSE */    ELSE,
    /* RID_WHILE */   WHILE,
50  /* RID_DO */      DO,
    /* RID_FOR */     FOR,
    /* RID_SWITCH */  SWITCH,
    /* RID_CASE */    CASE,
    /* RID_DEFAULT */ DEFAULT,
55  /* RID_BREAK */   BREAK,
    /* RID_CONTINUE */ CONTINUE,
    /* RID_RETURN */  RETURN,
    /* RID_GOTO */    GOTO,
    /* RID_SIZEOF */  SIZEOF,
60
    /* C 확장들
       C extensions */
    /* RID_ASM */     ASM_KEYWORD,
    /* RID_TYPEOF */  TYPEOF,
65  /* RID_ALIGNOF */ ALIGNOF,
    /* RID_ATTRIBUTE */ ATTRIBUTE,
    /* RID_VA_ARG */  VA_ARG,
    /* RID_EXTENSION */ EXTENSION,
    /* RID_IMAGPART */ IMAGPART,
70  /* RID_REALPART */ REALPART,
    /* RID_LABEL */   LABEL,
    /* RID_PTRBASE */ PTR_BASE,
    /* RID_PTREXTENT */ PTR_EXTENT,
    /* RID_PTRVALUE */ PTR_VALUE,
75
    /* RID_CHOOSE_EXPR */          CHOOSE_EXPR,
    /* RID_TYPES_COMPATIBLE_P */   TYPES_COMPATIBLE_P,

    /* RID_FUNCTION_NAME */        STRING_FUNC_NAME,
80  /* RID_PRETTY_FUNCTION_NAME */ STRING_FUNC_NAME,
    /* RID_C99_FUNCTION_NAME */    VAR_FUNC_NAME,

    /* C++ */
    /* RID_BOOL */    TYPESPEC,
85  /* RID_WCHAR */   0,
    /* RID_CLASS */   0,
    /* RID_PUBLIC */  0,
    /* RID_PRIVATE */ 0,
    /* RID_PROTECTED */ 0,
90  /* RID_TEMPLATE */ 0,
```

```
    /* RID_NULL */     0,
    /* RID_CATCH */    0,
    /* RID_DELETE */   0,
    /* RID_FALSE */    0,
95  /* RID_NAMESPACE */ 0,
    /* RID_NEW */      0,
    /* RID_OPERATOR */ 0,
    /* RID_THIS */     0,
    /* RID_THROW */    0,
100 /* RID_TRUE */     0,
    /* RID_TRY */      0,
    /* RID_TYPENAME */ 0,
    /* RID_TYPEID */   0,
    /* RID_USING */    0,
105
    /* Cast 들
      casts */
    /* RID_CONSTCAST */ 0,
    /* RID_DYNCAST */ 0,
110 /* RID_REINTCAST */ 0,
    /* RID_STATCAST */ 0,

    /* 동일한 철자들
      alternate spellings */
115 /* RID_AND */      0,
    /* RID_AND_EQ */   0,
    /* RID_NOT */      0,
    /* RID_NOT_EQ */   0,
    /* RID_OR */       0,
120 /* RID_OR_EQ */    0,
    /* RID_XOR */      0,
    /* RID_XOR_EQ */   0,
    /* RID_BITAND */   0,
    /* RID_BITOR */    0,
125 /* RID_COMPL */    0,

    /* Objective C */
    /* RID_ID */             OBJECTNAME,
    /* RID_AT_ENCODE */      ENCODE,
130 /* RID_AT_END */         END,
    /* RID_AT_CLASS */       CLASS,
    /* RID_AT_ALIAS */       ALIAS,
    /* RID_AT_DEFS */        DEFS,
    /* RID_AT_PRIVATE */     PRIVATE,
135 /* RID_AT_PROTECTED */   PROTECTED,
    /* RID_AT_PUBLIC */      PUBLIC,
    /* RID_AT_PROTOCOL */    PROTOCOL,
    /* RID_AT_SELECTOR */    SELECTOR,
    /* RID_AT_INTERFACE */   INTERFACE,
140 /* RID_AT_IMPLEMENTATION */ IMPLEMENTATION
  };
```

만약 이 enum RID 열거자를 수정하고자 한다면 모든 type modifier 들은 시작시 하나의 block 내에 반드시 있어야 하는데 그것은 mask bit 로써 사용되기 때문이다. 27 개의 type modifier 들이 있으며 만약 우리가 좀 더 추가시킨다면 mask mechanism 을 반드시 재설계 해야 할 것이다.

### 3.2 예약어의 등록

이 예약어의 등록은 $prefix/gcc/c-parse.in 에 선언되어 있는 init_reswords () 함수에서 처리하게 된다. 즉 위에서 나열한 전역 변수 reswords 의 각각을 $prefix/gcc/c-common.c 에 선언되어 있는 전역 변수 ridpointers 에 넣게 된다.

```
tree *ridpointers;
```

위와 같은 원형을 가진 전역 변수 ridpointers 의 element 들은 예약된 type 이름들과 storage class 들을 위한 identifier node 들입니다. 그것은 RID_.... 값에 따라 나열되어 있다. 아래와 같은 pseudo 코드로 표현할 수 있을 것이다.

```
함수 init_reswords
{
  옵션에 따른 mask 비교

  ridpointers 공간 할당
  for (예약어 숫자 크기 만큼)
    {
      만약 (mask 와 비활성화가 설정되어 있다면)
        넘어감

      get_identifier () 함수를 통해 예약어를 ident_hash 에 등록
      매크로 C_RID_CODE 로 RID code 등록
      매크로 C_IS_RESERVED_WORD 로 1 등록
      ridpointers 에 RID code 번호대로 넣음
    }
}
```

## 제 4 절 yylex () 함수

GCC 에서는 LALR(1) 문법을 사용하기 위해서 bison 을 사용하고 있으며 LALR(1) 문법이 다음 status 로 전진하기 위해 읽어들이는 token 은 $prefix/gcc/c-parse.in 내에 선언되어 있는 yylex () 함수에 의해 처리된다.

전체적인 yylex 의 모습은 얇은 wrapper 로써, 실질적인 부분은 _yylex () 함수에서 처리한다. 위에서 언급했듯이, yacc 용 token 과 C 용 token 이 엄연히 존재하는데, _yylex () 함수의 역활은 GCC 에서 해석한 C 용 token 을 yacc 용 token 으로 변환해서 yacc 에게 던져주는 것이다. 그리고 yacc 의 stack 에 쌓이게 되는 yylval 에게 적당한 값을 설정해 주는 것이다.

실질적으로 C 언어 소스는 $prefix/gcc/c-lex 파일에 선언되어 있는 c_lex () 내에서 모두 이루어지게 되며, 각 identifier 를 string pool 에 저장하거나, 모두 tree 구조를 만드는 것, 즉, yacc 에게 status 가 전달되기 전에 이루어져야 하는 모든 수행을 여기서 하게 된다.

여기에서 주의 깊게 다루어야 할 부분은 CPP_NAME 으로 분류된 것들인데, 예약어 (예를 들면, if, for, while 등등) 나 실제 identifier 들이 모두 이 type 으로 분류되기 때문이다. 그리고 실제 그에 맞는 yacc 용 token 은 yylexname () 함수가 처리하여 반환하게 된다.

yylexname () 함수를 살펴보면 아래와 같이 두가지 상황으로 구분할 수 있다.

1. 예약어일 경우
   - STRING_FUNC_NAME 인 경우
   - STRING_FUNC_NAME 가 아닌 경우
2. 예약어가 아닐 경우

예약어일 경우, 아래에서 다루게 되겠지만 c_lex () 함수는 yylval 의 ttype 를 설정하게 되는데, 이를 C_IS_RESERVED_WORD 매크로로 실제 확인 여부를 판가름한다.

만약 해석한 token 이 STRING_FUNC_NAME 인 경우가 존재할 수 있는데, 이 경우가 발생할 수 있는 상황은 "__FUNCTION__" 나 "__PRETTY_FUNCTION__" 와 같은 예약어를 사용하여 code 를 작성한 경우 이다.

STRING_FUNC_NAME 가 아닌 경우, RID code 에 따라 전역변수 ridpointers 에 의해 yylval.ttype 가 새로이 설정이 되며, RID code 를 yacc token 으로 변환한 값이 반환되어지게 된다.

예약어가 아닐 경우, lookup_name () 함수를 통해서 DECL 을 구한 후 그것이 TYPE_DECL 이 아닐 경우, IDENTIFIER 를, TYPE_DECL 일 경우, TYPENAME 을 반환하게 된다.

그럼 이제 다음 하위 섹션에서 c_lex () 함수를 이해하기 위해서는 무엇을 이해해야 하는지 살펴본 후 넘어가도록 하자.

## 4.1 c_lex () 함수

실질적인 GCC 의 C 언어 token 을 생성하는 곳은 이 함수에서 담당을 하고 있으며, 하나의 token 은 구조체 struct cpp_token 에 모든 정보가 들어가게 된다. 이 구조체에 대해서는 6 주 문서인 "기반 작업 (1) struct cpp_reader 란" 에 구조체에 대한 설명이 들어가 있기 때문에 그것을 참조하기 바란다.

이 함수는 호출되면서 yylval 의 구성 요소인 ttype 을 인자로 받아들이게 되는데, 이것의 C 용 token 의 type 에 따라 처리한 후 이 type 을 반환하는 것으로 모든 일을 마치게 된다. 즉 실질적인 하나의 token 을 처리하는 것은 cpp_get_token () 함수에게 맡긴다. 맡은 함수는 token 의 type 이 CPP_PADDING 가 될 때 까지 계속 loop 를 돌며 처리하게 되는 것이다. 이 함수에 대해서는 다음 섹션에서 알아보도록 하겠다.

c_lex () 함수의 주요 목적은 yylval 의 구성 요소인 ttype 을 제대로 설정하는 것이다. 부과적인 일로 전역변수 indent_level 를 설정하는 것도 있다. 그럼 cpp_get_token () 함수가 반환한 cpp_token *tok 의 type 에 따라 어떠한 수행을 하는지 알아보도록 하자.

- CPP_OPEN_BRACE

  전역 변수 indent_level 의 값을 1 증가 시킨다. 이 전역 변수는 C 언어에서 { 의 갯수에서 } 의 갯수를 뺀 수를 나타낸다.

- CPP_CLOSE_BRACE

  전역 변수 indent_level 의 값을 1 감소 시킨다.

- CPP_OTHER

  적당한 오류를 출력한 후 재시도를 하게 된다.

- CPP_NAME

  아래의 operation 을 수행하게 된다. 여기서 value 는 c_lex () 함수의 인자로 건네졌던 yylval 의 ttype 이다.

  ```
  *value = HT_IDENT_TO_GCC_IDENT (HT_NODE (tok->val.node));
  ```

- CPP_NUMBER

  아래의 operation 을 수행하게 된다.

  ```
  *value = lex_number ((const char *)tok->val.str.text, tok->val.str.len);
  ```

- CPP_CHAR 혹은 CPP_WCHAR

  아래의 operation 을 수행하게 된다.

  ```
  *value = lex_charconst (tok);
  ```

- CPP_STRING 혹은 CPP_WSTRING

아래의 operation 을 수행하게 된다.

```
*value = lex_string (tok->val.str.text, tok->val.str.len,
                     tok->type == CPP_WSTRING);
```

### 4.1.1 lex_number

cpp_get_token () 함수에서 어떠한 수행을 했다고 가정하고, 여기까지 도달하였다면 위에서 언급한 cpp_get_token () 함수를 모두 수행하여 cpp_token 내의 모든 값을 제대로 넣었음을 의미한다. 이제 yacc 가 사용하는 yylval 의 구성요소인 ttype 을 제대로 설정하기만 하면 되는 것이다.

cpp_token 의 type 이 CPP_NUMBER 로 해석된 것은 lex_number () 함수를 통해서 TREE 가 만들어 진다.

C 언어에서는 숫자의 형태는 여러 가지가 나열될 수 있는데, 대부분 10 진수 혹은 16 진수를 취할 것이다. 그에 대한 값 또한 각 머신이 담을 수 있는 용량의 크기를 맞춰져야 할 것이다. 그럼 lex_number () 함수는 숫자를 어떻게 담아 내고 있는지 살펴보도록 하자.

```
#define TOTAL_PARTS ((HOST_BITS_PER_WIDE_INT / HOST_BITS_PER_CHAR) * 2)
  unsigned int parts[TOTAL_PARTS];
```

> 우리는 실제로 각 part 내에 오직 HOST_BITS_PER_CHAR bit 들만 저장한다. Part 배열을 채울 아래 code 는 host int 는 host char 보다 최소 두배이상 크기이고, HOST_BITS_PER_WIDE_INT 는 HOST_BITS_PER_CHAR 의 짝수 배이라고 가정한다. 두 HOST_WIDE_INT 들은 우리가 저장할 수 있는 가장 큰 int literal 이다. 아래에 overflow 를 감지하기 위해서, Part 들 (TOTAL_PARTS) 의 개수는 두 HOST_WIDE_INT 들의 bit 들을 잡고 있는데 필요한 정확한 part 들의 number 여야만 한다.

이 변수를 통해서 일단 가상적으로 넣게 되는데, 위의 설명에서 알 수 있듯이 각 배열 요소 마다 HOST_BITS_PER_CHAR 만큼만 저장을 하게 되어 있다. 일반적인 펜티엄4 컴퓨터에서는 8 개의 배열이 할당된다. 이 배열내에 독립적으로 값을 저장해 놓아서 다른 머신들간에 있을 수 있는 unsigned int 나, unsigned char 크기의 이질성을 없앴으며, 나중에 실제 TREE 형성할 때는 그 값이 정수인지, 실수인지를 구분하여 node 가 생성되어 진다.

그럼 정수를 만들 때를 알아보도록 하자. 실제 정수를 위한 TREE node 는 HOST_WIDE_INT 크기로, high 와 low 를 아래의 루틴을 사용하여 만들게 된다.

```
for (i = 0; i < HOST_BITS_PER_WIDE_INT / HOST_BITS_PER_CHAR; i++)
  {
    high |= ((HOST_WIDE_INT) parts[i + (HOST_BITS_PER_WIDE_INT
                                        / HOST_BITS_PER_CHAR)]
             << (i * HOST_BITS_PER_CHAR));
    low |= (HOST_WIDE_INT) parts[i] << (i * HOST_BITS_PER_CHAR);
  }
```

최종적으로 만들어지는 TREE 는 아래의 루틴을 통해서 GCC 가 해석한 숫자에 대한 실제 TREE node 가 만들어지게 된다.

```
value = build_int_2 (low, high);
```

최종적으로 숫자를 위한 node 가 생성되었다면, TREE_TYPE (value) 에 대해서만 설정해주면 완료가 되는 것이다. 여기서 type 을 고를 때는 잠시 생각해 봐야 할 것이 있는데, Traditional type 이 있고 ISO type 두 개가 존재한다는 것이다. 많은 영향을 주는 것은 아니지만, 어떤 type 을 사용하느냐에 따라서 정수형 node 가 가지는 type 이 달라지게 된다.

하지만 편하지 않는 것은 표준 C 정수 type 들 내에서 달라지는 것이다. 잠시 표준 C 정수 type 들에 대해서 언급하면 아래와 같이 선언되어 있다.

```
/* 표준 C 정수 type 들.  이 배열내의 순서는 integer_type_kind 를
   사용합니다.  */
extern tree integer_types[itk_none];

#define char_type_node                  integer_types[itk_char]
#define signed_char_type_node           integer_types[itk_signed_char]
#define unsigned_char_type_node         integer_types[itk_unsigned_char]
#define short_integer_type_node         integer_types[itk_short]
#define short_unsigned_type_node        integer_types[itk_unsigned_short]
#define integer_type_node               integer_types[itk_int]
#define unsigned_type_node              integer_types[itk_unsigned_int]
#define long_integer_type_node          integer_types[itk_long]
#define long_unsigned_type_node         integer_types[itk_unsigned_long]
#define long_long_integer_type_node     integer_types[itk_long_long]
#define long_long_unsigned_type_node    integer_types[itk_unsigned_long_long]
```

위 리스트내에 type 들 중 적당한 것이 선정되어 기록되게 되는데, 적당한 것을 찾는 루틴이 int_fits_type_p () 함수이다. 이 함수는 해당 value 에 대한 적당한 type 을 찾을 경우 true 를 반환한다.

그럼 정수형태가 아닌 지수형태나 실수 형식으로 번호가 들어오면 어떻게 될까? 우선적으로 전체적인 숫자의 원형을 copy 라는 변수에 넣습니다. 예를 들어 설명을 한다면, 아래와 같은 변수를 선언할 때, lex_number 가 지수를 해석해야 할 사항이 있을 때, copy 변수에 "0.23e-01" 를 복사해 높는다.

```
int we=0.23e-01;
```

그런후 숫자 뒤에 붙을 수 있는 suffix 에 대한 처리를 하게 된다. 숫자에 대한 정보가 값이 있을 수 있고, suffix 에 대한 기호도 있을 수 있기 때문에 단일화 하기 위해 아래와 같은 구조체를 사용해서 저장하게 된다.

```
struct pf_args
{
  /* Input */
  const char *str;
  int fflag;
  int lflag;
  int base;
  /* Output */
  int conversion_errno;
  REAL_VALUE_TYPE value;
  tree type;
};
```

각각에 대해 잠시 설명을 한다면 아래와 같다.

- str

  위에서 언급한 copy 변수의 포인터가 여기 저장된다.

- fflag

  'f' 혹은 'F' suffix 를 위한 것이다. 이 suffix 가 설정될 경우 1 로 초기화된다.

- lflag

  'l' 혹은 'L' suffix 를 위한 것으로 설정된 경우 1 로 설정된다.

- base

지금 해석하고 있는 숫자의 진수를 말한다. 숫자의 형태가 10 진수이면, 값 10 이 저장되어 있고, 16 진수이면 값 16 이 저장되어 있다.

- conversion_errno

  실제 decimal-to-binary 변환을 하는 동안 발생한 오류 번호. 즉 문자열로 구성된 숫자를 실제 REAL_VALUE_TYPE 으로 변경하면서 발생한 errno 를 말한다. 참고적으로 말하지만 REAL_VALUE_TYPE 는 필자가 사용하는 컴퓨터에서는 int[5] 로 구성된 배열이다.

- value

  실제 decimal-to-binary 변환을 통해 변화된 output 값. 여기에 컴퓨터가 알아보기 좋게 변형된 값이 실제 존재한다. 이것이 나중에 TREE node 를 구성하는데 반영된다.

- type

  이 값이 가져야 하는 TREE node 의 type 을 결정한다. 기본적으로 "double" 형을 추천하고 있으나, 구성요소인 fflag 가 1 로 설정되어 있을 경우 "float", lflag 가 설정되어 있을 경우 "long double", 전역 변수 flag_single_precision_constant 가 설정되어 있을 경우, "float" 가 설정된다.

실제 Output 을 REAL_VALUE_TYPE 에 담는 작업은 아래의 루틴이 맏는다.

```
if (args->base == 16)
  args->value = REAL_VALUE_HTOF (args->str, TYPE_MODE (args->type));
else
  args->value = REAL_VALUE_ATOF (args->str, TYPE_MODE (args->type));
```

이제 위를 통해 수집한 정보를 통해서 실제 TREE node 를 구성한다. 실제 TREE node 구성은 아래의 루틴에서 맏는다.

```
if (imag)
  value = build_complex (NULL_TREE, convert (type, integer_zero_node),
                         build_real (type, real));
else
  value = build_real (type, real);
```

이 루틴에서 imag 라는 변수는 'i' 혹은 'I' suffix 가 숫자뒤에 붙어있으면, 1 로 설정된다. 즉 복소수 생성이 필요할 경우에만 build_complex () 함수가 호출되며, 실수 생성시에는 build_real () 함수를 호출하게 된다.

### 4.1.2 lex_charconst

이 것은 CPP_CHAR 혹은 CPP_WCHAR 와 같은 token 이 인식되었을 때 호출되는 것으로 아래와 같이 '문자' 가 선언될 때 호출된다고 보면 된다.

```
int we='a';
```

이 함수는 복작한 일을 하지 않으며, 문자열로 구성되어 있는 문자를 HOST_WIDE_INT 로 변환시키는 것이 모든 일이다. 이 문자의 TREE node 의 type 에 대해서도 생각해 봐야 하는데, C 에서 문자 상수는 type 'int' 를 가지며, C++ 에서는 'char' 를 가지고 있지만 multi-char charconst 들은 type 'int' 를 가지고 있다. build_int_2 () 함수를 통해서 기본적인 TREE node 를 만들고 마지막으로 적당한 type 을 넣어 주게 된다.

### 4.1.3 lex_string

CPP_STRING 혹은 CPP_WSTRING 이 해석되었을 때, 이에 맞는 TREE node 를 생성하기 위해 이 함수를 사용하는데, 이 함수의 역활은 중요한 것만 역는 다면, 간단해 진다. build_string () 함수를 통해서 TREE 에 대한 대부분의 기본 설정된 node 를 만든 후 이것이 CPP_STRING 일 때는 type 으로 char_array_type_node 를, CPP_WSTRING 일 경우 wchar_array_type_node 를 설정하는 것이 전부이다.

## 4.2 cpp_get_token () 함수

이 함수를 호출하게 되는 c_lex () 함수는 실제로 C 소스로 부터 token 을 하나 하나 읽어서 해석하지 않고, CPP_PADDING 를 기준으로 처리하게 되는데, 여기서 CPP_PADDING 는 C 소스의 white space 를 가르키는 것으로 공백이나, tab 문자들이 여기에 해당될 수 있다. 공백으로 나눈 후, 나머지 부분에 대해서는 이 섹션에서 언급할 cpp_get_token () 함수가 처리하게 된다. 이 함수의 역활은 cpp_token 구조체를 완성시키는 것이다.

context→prev 가 NULL 일 경우, _cpp_lex_token () 함수를 호출하게 된다. 대부분의 경우 이 함수가 호출되게 된다. 이 함수에서는 token 을 담을 공간, 즉 tokenrun 이 부족할 경우 채우고, 다시 _cpp_lex_direct () 함수로 넘기게 된다.

_cpp_lex_token () 함수에서 또한 C 언어에서 사용되는 macro 에 대해서도 처리하게 되는데, 이에 대해서는 다른 섹션에서 언급하다록 하겠다.

_cpp_lex_direct () 함수부터 본격적으로 각 token 의 첫번째 글자에 따라 어떻게 처리되느냐가 구분되어진다. 우선 그 구분에 들어가기 앞서 잠시 언급해야 할 부분이 있는데, 세가지 정도이다.

첫째, 이 함수의 결과로 반환되는 result 의 주소는 pfile→cur_token++ 의 값이다. 아래의 섹션 "Token 의 관리" 를 보면 나오겠지만, tokenrun 을 통해 관리되기 때문에 이러한 operation 을 수행할 수 있다.

둘째, 실제 C 소스는 pfile→buffer 내에 존재한다. 실제 그 token 의 첫번째 문자를 확인하기 위해서 아래와 같이 접근하여 읽는다.

```
buffer = pfile->buffer;
c = *buffer->cur++;
```

이 변수 c 값에 따라 처리되는 구분이 달라지는 것이다.

셋째, 결과로 반환되는 result→flags 값은 buffer→saved_flags 의 값에 영향을 받는다는 것이다. 새로이 result 공간을 할당 받은 후, result→flags 값이 이 함수의 처음 부분에서 buffer→saved_flags 로 설정이 되어진다.

이제 본격적으로 각 c 값에 따라 처리 과정을 보게 될텐데, 그 처리에 앞서 우리에게 중요한 것은 이 함수로부터 반환되는 result 의 구성요소가 어떻게 설정되는지가 가장 중요하기 때문에 이 점을 중점적으로 보며, 그 외 다른 영향을 미치지 않는 부분은 skip 할 것이다.

- 공백문자 혹은 \0

  c 값이 공백 문자일 경우, 전부 skip 하게 된다. 만약 c 값이 '\0' 를 나타낼 경우, 결과값의 type 은 CPP_EOF 가 된다.

- \n 혹은 \r

  GCC 가 관리하는 행, 열에 대해 새로이 정리하고, 그 줄이 가지는 특징, 즉 cpp_buffer 의 saved_flags 들을 새로이 설정해 준다. 만약 이 줄의 의미가 directive 가 아닐 경우, pfile→keep_token 이 설정되어 있지 않으면, tokenrun 을 새로이 설정하고 goto 문으로 새시작한다. 만약 이 줄의 의미가 directive 일 경우, 결과값의 type 을 CPP_EOF 로 설정하고 끝마친다.

- '?' 혹은 \\

  C 언어에서 \\ 의 의미는 여러 줄에 걸쳐 statement 를 작성할 때 사용하기 때문에 중요하지 않고, '?' 가 올 경우 결과값의 type 을 CPP_QUERY 로 설정한다.

- 숫자 (0 부터 9 까지)

  결과값의 type 을 CPP_NUMBER 로 설정하고 아래의 구문을 수행한다.

  ```
  parse_number (pfile, &result->val.str, c, 0);
  ```

- 'L'

L 로 시작할 경우 특별한 의미를 가지고 있는데, wide character 들 혹은 문자열들의 시작 값일 수 있기 때문이다. 만약 wide character 들 혹은 문자열들의 시작 값일 경우, 해당 사항을 처리하고 반환된다. 이럴 경우 결과값의 type 은 CPP_WSTRING 혹은 CPP_WCHAR 로 설정되며, 문자열을 해석하기 위해 아래의 루틴이 호출된다.

```
parse_string (pfile, result, c);
```

하지만 주의해야 할 것이 있는데, 대문자 L 로 시작하는 identifier 도 존재할 수 있기 때문이다. 아래에서 identifier 에 대해 알아보겠다.

- 대소문자 [a-zA-Z_]

  결과값의 type 에 CPP_NAME 을 설정하고 아래의 루틴을 수행한다.

  ```
  result->val.node = parse_identifier (pfile);
  ```

  위의 루틴을 수행한 후 Named operator 들을 그들 적당한 type 들로 변환한다. Named operator 같은 경우 C++ 에서만 해당되기 때문에 고려할 필요는 없을 것이다.

- ‘\” 혹은 ‘”’

  문자열 혹은 문자의 시작임을 알 수 있다. 문자열일 경우 결과값의 type 은 CPP_STRING 으로, 문자일 경우 CPP_CHAR 가 설정되며, 아래의 루틴이 수행된다.

  ```
  parse_string (pfile, result, c);
  ```

- ‘/’

  이 문자는 여러 의미를 가질 수 있는 문자로써, GCC 에서는 아래와 같은 의미를 내포할 수 있다.

  /* 와 같은 C 언어 comment 의 시작 문자.
  　중요한 내용이 없으니, Skip.
  // 와 같은 C++ 언어 comment 의 시작 문자.
  　중요한 내용이 없으니, Skip.
  /= 와 같은 나누기 연산
  　결과값의 type 을 CPP_DIV_EQ 로 설정한다. 그외의 일은 하지 않는다.
  / 와 같은 나누기 연산
  　buffer→backup_to 에 있는 값을 buffer→cur 에 넣고 결과값의 type 을 CPP_DIV 로 설정한다. 그외의 일은 하지 않는다.

- ‘<’

  이 문자도 여러 의미를 가질 수 있는 문자로써, GCC 에서는 아래와 같은 의미를 내포할 수 있다.

  Directive 내에서의 의미일 경우
  　이 경우는 아래와 같이 include 구문을 해석할 때 성립될 수 있다.

  ```
  #include <stdio.h>
  ```

  　이 같은 경우 < 문자가 들어가기 때문인데, GCC 에서는 pfile→state.angled_headers 를 설정함으로써 현재의 위치를 알려준다.
  <= 일 경우
  　결과값의 type 을 CPP_LESS_EQ 로 설정하고 끝낸다.
  << 일 경우
  　결과값의 type 을 CPP_LSHIFT 로 설정하고 끝낸다.
  <<= 일 경우

결과값의 type 을 CPP_LSHIFT_EQ 로 설정하고 끝낸다.

<? 일 경우

결과값의 type 을 CPP_MIN 로 설정하고 끝낸다.

<? = 일 경우

결과값의 type 을 CPP_MIN_EQ 로 설정하고 끝낸다.

<: 일 경우

결과값의 flags 에 DIGRAPH 를 OR 연산해서 더하고 결과값의 type 을 CPP_OPEN_SQUARE 로 수정한다.

<% 일 경우

결과값의 flags 에 DIGRAPH 를 OR 연산해서 더하고 결과값의 type 을 CPP_OPEN_BRACE 로 수정한다.

< 일 경우

buffer→backup_to 에 있는 값을 buffer→cur 에 넣고 결과값의 type 을 CPP_LESS 로 설정한다. 그외의 일은 하지 않는다.

- ‘>’

이 문자도 여러 의미를 가질 수 있는 문자로써, GCC 에서는 아래와 같은 의미를 내포할 수 있다.

<= 일 경우

결과값의 type 을 CPP_GREATER_EQ 로 설정하고 끝낸다.

>> 일 경우

결과값의 type 을 CPP_RSHIFT 로 설정하고 끝낸다.

>>= 일 경우

결과값의 type 을 CPP_RSHIFT_EQ 로 설정하고 끝낸다.

>? 일 경우

결과값의 type 을 CPP_MAX 로 설정하고 끝낸다.

>? = 일 경우

결과값의 type 을 CPP_MAX_EQ 로 설정하고 끝낸다.

> 일 경우

buffer→backup_to 에 있는 값을 buffer→cur 에 넣고 결과값의 type 을 CPP_GREATER 로 설정한다. 그외의 일은 하지 않는다.

- ‘%’

%= 일 경우

결과값의 type 을 CPP_MOD_EQ 로 설정하고 끝낸다.

%: 일 경우

결과값의 flags 에 DIGRAPH 를 OR 연산해서 더하고 결과값의 type 을 CPP_HASH 로 수정하며, buffer→backup_to 에 있는 값을 buffer→cur 에 넣는다.

%:%: 일 경우

결과값의 flags 에 DIGRAPH 를 OR 연산해서 더하고 결과값의 type 을 CPP_PASTE 로 수정한다.

%> 일 경우

결과값의 flags 에 DIGRAPH 를 OR 연산해서 더하고 결과값의 type 을 CPP_CLOSE_BRACE 로 수정한다.

% 일 경우

buffer→backup_to 에 있는 값을 buffer→cur 에 넣고 결과값의 type 을 CPP_MOD 로 수정한다.

- ‘.’

  . 일 경우
  결과값의 type 을 CPP_DOT 로 설정하고 buffer→backup_to 에 있는 값을 buffer→cur 에 넣는다.
  ... 일 경우
  결과값의 type 을 CPP_ELLIPSIS 로 설정하고 끝낸다.
  .[0-9] 일 경우
  결과값의 type 을 CPP_NUMBER 로 설정하고, 아래와 같은 루틴을 수행한다.
  ```
  parse_number (pfile, &result->val.str, c, 1);
  ```
  .* 일 경우
  결과값의 type 을 CPP_DOT_STAR 로 설정하고 끝낸다.

- ‘+’

  ++ 일 경우
  결과값의 type 을 CPP_PLUS_PLUS 로 설정하고 끝낸다.
  += 일 경우
  결과값의 type 을 CPP_PLUS_EQ 로 설정하고 끝낸다.
  + 일 경우
  buffer→backup_to 에 있는 값을 buffer→cur 에 넣고, 결과값의 type 을 CPP_PLUS 로 설정하고 끝낸다.

- ‘−’

  − > 일 경우
  결과값의 type 을 CPP_DEREF 로 설정하고 끝낸다.
  − > ∗ 일 경우
  결과값의 type 을 CPP_DEREF_STAR 로 설정하고 끝낸다.
  −− 일 경우
  결과값의 type 을 CPP_MINUS_MINUS 로 설정하고 끝낸다.
  − = 일 경우
  결과값의 type 을 CPP_MINUS_EQ 로 설정하고 끝낸다.
  − 일 경우
  buffer→backup_to 에 있는 값을 buffer→cur 에 넣고, 결과값의 type 을 CPP_MINUS 로 설정하고 끝낸다.

- ‘&’

  && 일 경우
  결과값의 type 을 CPP_AND_AND 로 설정하고 끝낸다.
  &= 일 경우
  결과값의 type 을 CPP_AND_EQ 로 설정하고 끝낸다.
  & 일 경우
  buffer→backup_to 에 있는 값을 buffer→cur 에 넣고, 결과값의 type 을 CPP_AND 로 설정하고 끝낸다.

- ‘|’

|| 일 경우
  결과값의 type 을 CPP_OR_OR 로 설정하고 끝낸다.
| = 일 경우
  결과값의 type 을 CPP_OR_EQ 로 설정하고 끝낸다.
| 일 경우
  buffer→backup_to 에 있는 값을 buffer→cur 에 넣고, 결과값의 type 을 CPP_OR 로 설정하고 끝낸다.

- ':'

:: 일 경우
  결과값의 type 을 CPP_SCOPE 로 설정하고 끝낸다.
:> 일 경우
  결과값의 flags 에 DIGRAPH 를 OR 연산해서 더하고 결과값의 type 을 CPP_CLOSE_SQUARE 로 수정한다.
: 일 경우
  buffer→backup_to 에 있는 값을 buffer→cur 에 넣고, 결과값의 type 을 CPP_COLON 로 설정하고 끝낸다.

- '*'

*= 일 경우
  결과값의 type 을 CPP_MULT_EQ 로 설정하고 끝낸다.
* 일 경우
  결과값의 type 을 CPP_MULT 로 설정하고 끝낸다.

- '='

== 일 경우
  결과값의 type 을 CPP_EQ_EQ 로 설정하고 끝낸다.
= 일 경우
  결과값의 type 을 CPP_EQ 로 설정하고 끝낸다.

- '!'

!= 일 경우
  결과값의 type 을 CPP_NOT_EQ 로 설정하고 끝낸다.
! 일 경우
  결과값의 type 을 CPP_NOT 로 설정하고 끝낸다.

- '^'

^= 일 경우
  결과값의 type 을 CPP_XOR_EQ 로 설정하고 끝낸다.
^ 일 경우
  결과값의 type 을 CPP_XOR 로 설정하고 끝낸다.

- '#'

## 일 경우
  결과값의 type 을 CPP_PASTE 로 설정하고 끝낸다.
# 일 경우

결과값의 type 을 CPP_HASH 로 설정하고 끝낸다.

- '∼'

  결과값의 type 을 CPP_COMPL 로 설정하고 끝낸다.

- ','

  결과값의 type 을 CPP_COMMA 로 설정하고 끝낸다.

- '('

  결과값의 type 을 CPP_OPEN_PAREN 로 설정하고 끝낸다.

- ')'

  결과값의 type 을 CPP_CLOSE_PAREN 로 설정하고 끝낸다.

- '['

  결과값의 type 을 CPP_OPEN_SQUARE 로 설정하고 끝낸다.

- ']'

  결과값의 type 을 CPP_CLOSE_SQUARE 로 설정하고 끝낸다.

- '{'

  결과값의 type 을 CPP_OPEN_BRACE 로 설정하고 끝낸다.

- '}'

  결과값의 type 을 CPP_CLOSE_BRACE 로 설정하고 끝낸다.

- ';'

  결과값의 type 을 CPP_SEMICOLON 로 설정하고 끝낸다.

- '@'

  결과값의 type 을 CPP_ATSIGN 로 설정하고 끝낸다. @ 는 objective C 의 punctuator 이다.

- '$'

  만약 CPP_OPTION (pfile, dollars_in_ident) 가 설정되어 있다면, 즉 identifier 에 $ 문자를 사용하도록 허락한다면 identifier 로 해석될 것이다.

- 그외의 모든 문자

  결과값의 type 을 CPP_OTHER 로 설정하고 결과값의 val.c 에 지금 분류의 기준이 되는 문자 c 를 넣고 끝낸다.

### 4.3 Number 와 String, Identifier 의 처리

GCC 가 숫자를 만났을 때 어떻게 처리를 할까? (즉, 여기에서는 parse_number () 함수의 루틴을 설명했다.) 숫자를 처리한다는 것은 [0-9\.+\-eEpP] 를 처리한다는 것을 말한다. 그럼 GCC 가 해석한 숫자는 어디에 저장될까? 그에 대한 답을 찾을려면 전역 변수 parse_in 의 u_buff 구성요소를 살펴보면 답이 보일 것이다. 이것은 unaligned 형태를 가지는 문자열 저장소이다. GCC 에서는 숫자를 처리할 때 이곳에 저장을 하고 그에 대한 pointer 를 cpp_token 의 val.str 에 기록한다.

(parse_string () 함수에 대한 설명) 문자열 또한 전역 변수 parse_in 의 u_buff 구성요소에 저장됨을 확인할 수 있다. 하지만 문자열 처리 과정에서 눈에 뛰는 점을 꼽으라면, Trigraph 와 Digraph 를 처리하는 과정이 존재한다는 것이다. 그 외에도 #include directive 와 같은 것도 이 parse_string () 함수에서 처리된다는 점이다. 문자열도 마찬가지로 u_buff 에 저장하고 그에 대한 포인터를 val.str 에 기록한다.

위 전역 변수 parse_in 의 u_buff 구성요소에 대해서는 6 주 문서인 “기반 작업 (1) struct cpp_reader 란” 를 참조하라.

(parse_identifier () 함수에 대한 설명) GCC 가 identifier 를 만났을 경우는 다음과 같이 처리한다. 대단한 내용은 없고, 결과적으로 소스 코드가 있는 pfile→buffer→cur 를 읽어서, ISIDNUM 매크로가 참일 때까지 전진한 후, 해석한 이름을 기반으로 아래와 같은 루틴을 실행한다.

```
result = (cpp_hashnode *)
  ht_lookup (pfile->hash_table, base, cur - base, HT_ALLOC);
```

결과적으로 전역 변수 ident_hash 에 저장되게 되며, ht_lookup 함수가 반환한 result 는 cpp_token→val.node 에 들어가 반환되게 된다.

만약 identifier 가 여러줄에 걸쳐있거나, dollar($) 혹은 ? 기호가 들어가 있을 경우, parse_identifier_slow () 함수를 호출하게 된다.

## 제 5 절 Token 의 관리

GCC 에서는 수많은 token 들을 만나게 될텐데, 그들은 어떻게 이 많은 것을 관리하게 되는 걸까? 그것에 대해서는 struct tokenrun 구조체를 알아야 할 것이다. 이 구조체에 대해서는 6 주 문서인 “기반 작업 (1) struct cpp_reader 란” 에 구조체에 대한 설명이 들어가 있기 때문에 그것을 참조하기 바란다.

struct tokenrun 구조체는 GCC 가 읽어들이는 수많은 token 을 관리하는 구조체이다. 이 구조체는 struct cpp_reader *parse_in 내부에 선언되어 있는데, 그 중 base_run 구성요소는 전체적인 tokenrun 의 시작 부분을 가르키는 요소이다. 즉, 이 구성요소의 prev 나 next 가 생겨나더라도, 처음에 만들어 진 것이 이것이다라고 계속 가르키고 있다. cur_run 구성요소는 현재 활동중인 tokenrun 을 가르키고 있다. 대부분 250 개 정도의 token 을 위한 공간을 가지고 있으며, 이 공간이 꽉 다찼을 경우에, cur_run 은 새로운 값으로 변경되게 된다.

그럼 언제 이 공간이 꽉 다찼는지를 검사하는 것일까? 그 부분은 _cpp_lex_token 의 처음 부분에 기술되어 있다. 그리고 이 공간이 변화하는 분기점은 존재하지 않는가? 그 부분은 token 이 ‘\n’ 과 같은 newline 문자를 만났을 때 나타나며, _cpp_lex_direct () 함수에 보인다. 순서대로 다시 적으면 아래와 같이 요약할 수 있겠다.

- struct cpp_reader *parse_in 을 처음 만들 때, 처음으로 base_run 구성요소의 token 공간을 250 개 할당하고, cur_run 이 이것을 가르키게 설정한다. 그리고 cur_token 을 cur_run.base 로 가르키게 만들어 놓는다.

- 그런 다음에 token 을 열심히 해석한다. 각 token 에 따라 cur_run 의 공간을 cur_token 에 할당하는 부분은 _cpp_lex_direct 의 첫부분에 나와 있다.

- token 을 해석하다가 이 token 을 넣을 공간이 부족한지 검사를 하게 되는데, 그 부분이 _cpp_lex_token () 함수 부분이다. next_tokenrun () 함수를 사용하여 새로운 cur_run 용 token 공간을 만드는데, 이전에 쓰던 공간은 cur_run.prev 에 넣고, cur_token 을 새로 할당된 cur_run.base 에 넣음으로써 linked list 를 만든다.

- 다시 열심히 token 을 해석하며 쌓는다.

- _cpp_lex_direct () 함수에서 token 처리를 하다가 '\n' 혹은 '\r' 를 만나면, 지금 해석하는 statement 가 directive 가 아닐 경우, parse_in 구성요소 중 하나인 keep_tokens 를 살펴보게 되는데 만약 token 을 유지할 필요가 없으면 cur_run 을 base_run 으로 재설정하고 결과값을 담을 result 를 base_run.base 로 설정한다. 마지막으로 현재 처리하고 있는 token 을 가르키는 cur_token 값을 result + 1 값으로 설정하게 된다.

잠시 자세히 살펴봐야 할 부분도 있는데, parse_in 구성요소 중 하나인 keep_tokens 가 그것이다. token 을 해석하다보면 그냥 그 공간을 재사용할 수도 있는 경우가 있는데, 이 값이 1 이 아닐 경우, token 이 재사용되지 않는다.

# 제 6 절 Macro 의 처리

이 곳에서는 GCC 에서 directive 를 어떻게 처리하는지에 대해서 알아보도록 하자.

## 6.1 Directive 의 구성

GCC 에서 사용하는 directive 들은 매크로 DIRECTIVE_TABLE 에 자세히 설명되어 있다. 아래에서 자세한 정의를 볼 수 있다.

```
/* struct directive 의 기원 field 를 위한 값들.      KANDR directive 들은
   traditional (K&R) C 에서 유래되었습니다.         STDC89 directive 들은
   1989 C standard 에서 유래되었습니다.         EXTENSION directive 들은
   확장판입니다.          */
#define KANDR       0
#define STDC89      1
#define EXTENSION   2

/* struct directive 의 flags field 를 위한 값들.   COND 는 조건부를
   가르킵니다; IF_COND 는 opening conditional 를 의미.       INCL 는 ``...''
   와 <...> 를 각각 q-char 와 h-char 로 취급함을 의미합니다.            IN_I 는
   이 directive 가 -fpreprocessed 에 영향을 받더라고 다루어져야 함을
   의미합니다. (Callback hook 들을 가지는 directive 들이 존재합니다.)                */
#define COND        (1 << 0)
#define IF_COND     (1 << 1)
#define INCL        (1 << 2)
#define IN_I        (1 << 3)

/* 이것은 directive handler 들의 table 이다.      이것은 발생 빈도에 따라 정렬되어
   있으며, 끝 번호는 현재 우리가 소스 code 전체에서 사용되는 directive 의 수이다.
   (1999-05-18 날 egcs 와 libc CVS, 거기에 grub-0.5.91 와 linux-2.2.9,
   pcmcia-cs-3.0.9 를 더한다.)    이제 directive lookup 이 O(1) 라는 것은 별로
   중요하지 않다.          #warning 나 #include_next 와 같은 다른 확장들은 모두 빠져
   있다.    이름은 확장이 어디에서 나타나는지를 나타내고 있다.                    */

#define DIRECTIVE_TABLE                                              \
D(define,     T_DEFINE = 0, KANDR,    IN_I)         /* 270554 */ \
D(include,    T_INCLUDE,    KANDR,    INCL)         /* 52262 */  \
D(endif,      T_ENDIF,      KANDR,    COND)         /* 45855 */  \
D(ifdef,      T_IFDEF,      KANDR,    COND | IF_COND) /* 22000 */ \
D(if,         T_IF,         KANDR,    COND | IF_COND) /* 18162 */ \
D(else,       T_ELSE,       KANDR,    COND)         /*  9863 */  \
D(ifndef,     T_IFNDEF,     KANDR,    COND | IF_COND) /* 9675 */  \
D(undef,      T_UNDEF,      KANDR,    IN_I)         /*  4837 */  \
D(line,       T_LINE,       KANDR,    0)            /*  2465 */  \
D(elif,       T_ELIF,       STDC89,   COND)         /*   610 */  \
```

```
D(error,      T_ERROR,      STDC89,  0)           /*   475 */ \
D(pragma,     T_PRAGMA,     STDC89,  IN_I)        /*   195 */ \
D(warning,    T_WARNING,    EXTENSION, 0)         /*    22 */ \
D(include_next, T_INCLUDE_NEXT, EXTENSION, INCL) /*    19 */ \
D(ident,      T_IDENT,      EXTENSION, IN_I)      /*    11 */ \
D(import,     T_IMPORT,     EXTENSION, INCL)      /* 0 ObjC */ \
D(assert,     T_ASSERT,     EXTENSION, 0)         /* 0 SVR4 */ \
D(unassert,   T_UNASSERT,   EXTENSION, 0)         /* 0 SVR4 */ \
SCCS_ENTRY                                        /* 0 SVR4? */

/* #sccs 는 항상 인식되지는 않는다.               */
#ifdef SCCS_DIRECTIVE
# define SCCS_ENTRY D(sccs, T_SCCS, EXTENSION, 0)
#else
# define SCCS_ENTRY /* nothing */
#endif
```

마지막에 볼 수 있는 SCCS_ENTRY 는 #sccs directive 를 나타내는데, 이것은 항상 사용되는 것이 아니기 때문에, 머신마다 달라질 수 있다.

그럼 이 매크로를 어떻게 사용할까? GCC 에서는 이 매크로를 이용해서 필요한 여러 정보를 생성한다. 그 중 여러 정보를 담을 구조체 또한 제공하는데, struct directive 가 그것이며 아래에 나열하였다.

```
typedef void (*directive_handler) PARAMS ((cpp_reader *));
typedef struct directive directive;
struct directive
{
  directive_handler handler;
  const U_CHAR *name;
  unsigned short length;
  unsigned char origin;
  unsigned char flags;
};
```

이 구조체는 하나의 #-directive 를 정의하며 그것을 어떻게 다룰 것인지에 대한 내용도 포함한다. 우선 각 구성요소의 기능부터 아래에 나열하도록 하겠다.

- handler

  Directive 를 다룰 함수.

- name

  Directive 의 이름.

- length

  이름의 길이.

- origin

  Directive 의 기원

- flags

  이 Directive 를 표현하는 기호

이 구조체를 이용하여 전역 변수 dtable 에 각각에 대한 기록을 하게 되며, 전역 변 dtable 은 다음과 같은 매크로의 작동을 통해 설정된다.

```
#define D(name, t, origin, flags) \
{ CONCAT2(do_,name), (const U_CHAR *) STRINGX(name), \
  sizeof STRINGX(name) - 1, origin, flags },
static const directive dtable[] =
{
DIRECTIVE_TABLE
};
#undef D
#undef DIRECTIVE_TABLE
```

위의 내용을 보면 CONCAT2 매크로를 내부에서 사용하고 있음을 알 수 있는데, 각 매크로 이름에 do_.... 이 붙이는 것을 의미하며, 예를 든다면, define directive 의 경우, CONCAT2 매크로에 의해 do_define 라는 이름으로 설정되게 된다. 의미를 따진다면, do_.... 들은 모두 handler 로써 실제 각 directive 를 만났을 때 처리할 함수 들이다. 또한 이 handler 에 대한 선언도 해줘야 할 것이다. 이 함수는 아래와 같이 선언한다.

```
#define D(name, t, o, f) static void CONCAT2(do_,name) PARAMS ((cpp_reader *));
DIRECTIVE_TABLE
#undef D
```

또한 각 directive 에 대한 이름을 열거자 (enum) 로 사용하기 위해 아래와 같이 선언하기도 한다.

```
#define D(n, tag, o, f) tag,
enum
{
  DIRECTIVE_TABLE
  N_DIRECTIVES
};
#undef D
```

## 6.2 Directive 의 등록

이제 위에서 언급한 directive 구성원들은 실제 Lexer 가 C 소스를 처리하면서 발견하게 되면 이를 처리할 수 있도록 만들어줘야 한다. 이것은 "언어 의존적 초기화"에서 이루어지게 되는데, 지금까지의 문서에서는 이 부분에 대해서는 언급한 적은 없었다. 미리 이 macro 처리 부분에 대해서만 언급한다면 실제 등록은 _cpp_init_directives () 함수에 의해서 이루어지게 된다.

간단한 함수이며, 아래와 같은 과정이 모두 다이다.

```
void
_cpp_init_directives (pfile)
     cpp_reader *pfile;
{
  unsigned int i;
  cpp_hashnode *node;

  for (i = 0; i < (unsigned int) N_DIRECTIVES; i++)
    {
      node = cpp_lookup (pfile, dtable[i].name, dtable[i].length);
      node->directive_index = i + 1;
    }
}
```

물론 다시 언급하지 않아도 되겠지만, 다시 말하면 cpp_lookup () 함수를 통해서 이름을 등록할 경우, 결과적으로 String Pool 을 담당하는 전역 변수 ident_hash 에 저장될 것이다. 여기서 살펴봐야 할 것은 directive_index 를 설정한다는 것이다.

## 6.3 처리한 directive 저장소들

이 하위 섹션에서는 directive 를 처리할 때 그 directive 의 모든 정보를 저장할 때 사용하는 구조체와 실제 GCC 소스에서 처리한 매크로를 적용할 때 사용하는 구조체에 대해 알아보도록 하자. GCC 에서는 struct cpp_macro 구조체를 사용하여 각 directive 에 대한 정보를 저장하게 되는데, 자세한 사항은 아래와 같다.

```
struct cpp_macro
{
  cpp_hashnode **params;
  cpp_token *expansion;
  unsigned int line;
  unsigned int count;
  unsigned short paramc;
  unsigned int fun_like : 1;
  unsigned int variadic : 1;
  unsigned int syshdr   : 1;
};
```

각 요소에 대해서 설명은 하면 아래와 같다.

- params

  이 선언된 directive 의 parameter 들을 가지고 있다.

- expansion

  Replacement list 의 처음 token 을 가지고 있는데, 여기서 replacement list 란

  ```
  \#define foo(a) int a, b;
  ```

  라고 선언하였다면, “int a, b;”를 가지게 된다는 것이다.

- line

  이 directive 가 선언된 줄 번호

- count

  Replacement list 의 (즉, expansion 에서의) token 개수.

- paramc

  Parameter 들의 갯수.

- fun_like

  만약 function-like macro 라면 설정.

- variadic

  만약 variadic macro 라면 설정.

- syshdr

  만약 system header 에 정의된 macro 라면 설정.

이 구조체에 directive 에 대한 모든 정보를 기록하게 된다. 그럼 실제로 이 구조체가 어떻게 사용되는지에 대해서는 다음 섹션부터 언급될 것이다.

이제 GCC 에 실제 매크로가 적용되었을 때 그에 대한 내용을 적용하는데, 사용되는 구조체에 대해 알아보도록 하자. 아래에서 살펴볼 구조체는 함수 형태로 선언되어 있는 매크로나 다른 것들에서 볼 수 있는 argument 들을 담는데, 사용되는 것이다. 예를 들어 설명한다면, 아래와 같은 예제가 주어졌을 때

```
#define foo(a, b) (a > b)
foo(1, 2)
```

위의 두번째 줄에서 '(' 와 ')' 사이에 있는 1 과 2 를 저장하는 것을 말한다. 이 구조체의 원형은 아래와 같이 선언되어 있다.

```
typedef struct macro_arg macro_arg;
struct macro_arg
{
  const cpp_token **first;
  const cpp_token **expanded;
  const cpp_token *stringified;
  unsigned int count;
  unsigned int expanded_count;
};
```

각각에 대한 세부 설명은 아래와 같이 정의되어 있다.

- first

  Unexpanded argument 내 첫번째 token.

- expanded

  Macro-expanded argument.

- stringified

  Stringified argument.

- count

  Argument 내 token 들의 #.

- expanded_count

  Expanded argument 내 token 들의 #.

## 6.4 Directive 의 처리

지금까지 GCC 에서 사용하는 directive 에 무엇이 있으며, 어디에서 등록하는지 그리고 어디에 저장되는지 알아보았다. 이제 실제로 GCC 에서는 이 directive 를 인식하면 어떻게 행동하는지 알아보도록 한다.

C 언어에서는 Lexer 가 directive 를 만났을 경우, 그에 대해 하나씩 통채로 처리를 하게 된다. 각 token 에 대해서 yacc 에게 돌려주어 yacc 가 처리하도록 하는 것이 아니라, # (CPP_HASH) token 을 만났을 경우, 그와 관련된 모든 token 들을 처리한 후 넘어가게 된다.

C 언어에서는 이것이 directive 인지 아닌지를 구분하는 기준이 # 기호 이며, GCC 가 이 # 기호를 해석할 경우 반환활 cpp_token 의 type 으로 CPP_HASH 를 설정하게 된다. 하지만 이 CPP_HASH 기호는 바로 yacc 에 반영되지는 않고 _cpp_lex_token () 함수에 선언되어 있는 _cpp_handle_directive () 함수에 의해 처리가 완료된 뒤 계속 처리되게 된다.

말로 설명하기에는 한계가 있으니, 예제를 통해서 설명하도록 하겠다.

```
#define foo(a) b
```

위와 같은 문장을 처리한다고 생각하면, 아래와 같이 나열할 수 있겠다.

1. 우선 하나의 token 을 처리하기 위해서 cpp_get_token () 함수가 호출될 것이다.
2. 우선 _cpp_lex_token () 함수는 다시 _cpp_lex_direct () 함수를 호출할 것이다.

3. 이 함수를 통해서 첫번째 token 이 # 임을 인식하고 CPP_HASH 라고 설정한 후 자신을 호출한 함수에게 cpp_token 결과를 반환한다.

4. _cpp_lex_token () 함수에서는 현재 처리하고 있는 token 이 directive 라는 사실을 알았기 때문에, _cpp_handle_directive () 함수를 통해서 계속 처리하도록 호출한다.

5. _cpp_lex_token () 함수는 다시 _cpp_lex_token () 함수를 호출하여 # 다음에 있는 token 을 읽어 들인다. 이 함수에서 ident_hash 에 들어 있는 "define" identifier 에 대한 cpp_hashnode 를 얻게 된다. 그리고 이미 위해서 해당 cpp_hashnode→directive_index 에 "언어 독립적 초기화"에서 번호를 초기화 해놓았기 때문에 이를 이용하여 해당 directive 에 대한 handler 를 구할 수 있다.

6. cpp_hashnode→directive_index 를 통해 해당 directive 의 handler 를 전역변수 dtable 에서 구한다.

7. 인식한 struct directive 의 값을 parse_in→directive 에 넣고, 실제 handler 인 do_define () 함수를 수행한다.

## 6.5 Directive handler

위의 예제를 계속 설명하도록 하자. #define 구문의 경우 do_define () handler 함수가 수행되는데, 이곳에 우선 lex_macro_node () 함수를 통해 define 의 이름을 얻게 된다. 이에 대한 이름이 있을 경우, 실제로 _cpp_create_definition () 함수를 이용하여 이에 대한 정보를 만들게 되며, _cpp_create_definition () 함수는 cpp_macro 에 정보를 저장하게 된다. 실제로 cpp_macro 를 새로이 생성하고 그에 대한 정보를 넣는 것은 모두 이 함수에서 수행하게 된다.

결과적으로 그럼 cpp_macro 구조체는 어디에 들어가는 것일까? do_define () 함수에서는 해당 #define 이 C 언어에서 사용될 이름에 대한 node 를 구하였는데, 그 node 를 _cpp_create_definition () 함수에게 건내었다. 즉 cpp_macro 구조체는 이 node 에 들어가게 된다. 어떻게 어떤 순서로 들어가는지는 이제 알아 보도록 하자.

아래는 _cpp_create_definition () 함수의 수행과정이다.

1. 우선 cpp_macro 를 하나 할당한다. 이 구조체에 대해서는 위의 "처리한 directive 저장소" 섹션을 참조하기 바란다. 새로이 할당을 한 cpp_macro 에 구성요소 line 을 설정한다.

2. 이제부터 실제 매크로 이름 foo 다음부터 계속해서 token 을 해석해 나가야 한다. 위의 예제에서는 "(a) b"가 남아 있으므로, '(' 토큰부터 해석될 것이다. 여기서 언급해야 할 것은 '(' 와 ')' 사이에 존재하는 것은 parameter 들이라는 것이다. 결과적으로 이 parameter 들은 cpp_macro 의 params 에 들어가게 되고, 그 parameter 들의 숫자가 paramc 에 들어가게 된다. 그럼 이 paramter 들은 어디에 저장되는지가 문제인데, 이 parameter 들은 parse_in→a_buff 에 순서대로 저장된다.

3. '(' 와 ')' 사이에 parameter 들이 존재하기 때문에, 이 define 구문은 함수 형태이기에 cpp_macro 구성요소인 fun_like 가 1 로 설정된다.

4. 이제 "b" 를 해석해야 할 차례이다. 이 정보는 실제 나중에 C 언어에서 foo(LLL) 이라고 선언하였을 때, "b" 로 변화해야 때문에, 이 정보들은 모두 expansion 에 들어가야 할 것이다. 여기서 해석된 모든 token 또한 parse_in→a_buff 에 순서대로 저장되게 되며, token 의 숫자가 구성요소인 count 에 들어가게 된다.

5. 이제 cpp_macro 를 구성해야 할 요소들은 모두 들어가게 되었다. 이제 이 cpp_macro 를 define 매크로 이름 node 에 넣어야 하는데, 아래와 같이 수행해서 넣는다.

```
node->type = NT_MACRO;
node->value.macro = macro;
```

각 구성요소에 대한 정보는 "06-기반 작업 (1) struct cpp_reader 란" 문서를 참조하기 바란다.

## 6.6 처리한 directive 의 적용

이 하위 섹션에서는 위에서 처리하여 생성된 매크로에 대한 적용이 어떻게 이루어지는지 살펴보도록 하자. 아래에 위의 예제를 약간 수정한 새로운 예제를 보자.

```
#define foo(a) int a;
foo(b)
```

생성된 매크로의 이름은 foo 이다. #define 구문으로 사용하였기 때문에, 아래부터 foo 라는 identifier 를 만나면 'int a' 구문으로 변경을 해줘야 한다. 위의 예제에서는 foo(b) 를 선언되었기 때문에, 결과적으로는

```
int b;
```

위의 예제가 선언된 것과 마찬가지로 적용되는 것이다. 그럼 GCC 에는 이렇게 처리하기 위해 어떻게 하고 있는지 아래에서 살펴보도록 하자.

위에서는 foo 라는 매크로가 어떻게 생성되는지 보았다. 다음 두번째 줄에는 이 선언된 매크로를 선언한 구문이 나온다. 두번째 줄의 foo 를 만났을 때, GCC 에서 어떻게 행동하는지에 대해 조명을 해야 할 것이다.

1. Lexer 가 foo 를 읽게되는데, 이미 위에서 전역 변수 ident_hash 에 foo 를 #define 이 선언될 때, 만들어 놓았기 때문에, 해당 foo 에 대한 cpp_hashnode 를 얻게 된다. 이 node→type = NT_MACRO 이고, node→value.macro = macro 가 선언되었음을 위해서 말하였다.

2. 이제 이 결과값을 반환받은 cpp_get_token 함수에서는 이 node 를 확인 한 후 NT_MACRO 일 경우, enter_macro_context () 함수를 호출함으로써 매크로를 처리할 준비를 한다. 그럼 준비 과정은 어떻게 하는 것일까? 그에 대한 답은 cpp_context 구조체가 가지고 있다. 이 구조체의 설정은 "06-기반 작업 (1) struct cpp_reader 란" 문서를 참조하기 바라며, cpp_context 구조체내에 대체할 token 들의 list 넣고, 그에 대한 list 를 parse_in→context 에 넣음으로써 준비를 완료하게 된다.

3. 그럼 준비가 완료된 후, 어떤 과정을 거치는지에 대해 이야기를 한다면, cpp_get_token () 함수를 살펴봄으로써, 이해할 수 있다. 즉, 이 함수의 경우, parse_in→context 를 살펴보고 값이 존재할 경우, 안에 존재하는 token 들을 c_lex () 함수에게 보내게 된다. 값이 존재하지 않는다면, 일반적인 _cpp_lex_token () 함수가 호출되어 진다. cpp_context 구조체가 존재하는 이유는 directive 로 인해 내부 identifier 가 대체될 필요가 있을 경우, 대체될 token 들을 잡고 있기 위해서 이다.

4. 이제 모든 context 를 대체했을 경우, 그에 대한 context 를 free 시킴으로써 다음 token 들이 영향을 받지 않도록 한다.

위에서는 대략적인 과정에 대해서 나열을 하였기 때문에, 아래부터는 실제 cpp_context 를 채우는 과정을 수행하는 enter_macro_context () 함수에 대해 더 자세히 알아보도록 하자.

1. 이 함수까지 들어오게 될 경우, 위의 예제에서 언급했듯이 foo identifier 를 인식한 경우이며 이 identifier 의 node 의 type 은 NT_MACRO 이며, 해당 macro 에 대한 정보는 node→value.macro 에 들어가 있다.

2. 이제 이 macro 가 builtin 매크로인지 아닌지를 구분하는데, 위에서 정의한 foo 는 builtin 이 아니며, 함수 형식이기 때문에 실제 소스에 정의된 parameter 들을 해석하는 단계에 들어간다.

3. 위의 소스에서는 '(' 와 ')' 사이에 b 문자 하나만 들어가 있으며, 실제 b token 을 해석하는 함수는 funlike_invocation_p () 함수내에서 모두 수행하게 된다. 우선 이 함수에 돌입하기 전에 몇몇 state 들을 설정해 줄 필요가 있다.

   ```
   pfile->state.prevent_expansion++;
   pfile->keep_tokens++;
   pfile->state.parsing_args = 1;
   ```

각각의 의미는 "06-기반 작업 (1) struct cpp_reader 란" 문서를 참조하기 바라며, 실제로 '(' 와 ')' 사이의 token 을 모으기 시작한다. 그럼 이 parameter (여기에서 'b' 만 존재하는데) 들이 어떻게 정리되어 저장될 것인가? 아래 그림 1 를 참조하면 이해하기 빠를 것이다. Parameter 의 수 만큼 macro_arg 구조체를 _cpp_buff 에 할당하고, 각 parameter 마다 가지고 있는 token 의 수 만큼 macro_arg→first[...] 에 넣게 된다. 물론 macro_arg→first[...] 를 위한 공간 또한 앞에서 사용한 _cpp_buff 와 같은 곳에 위치한다.

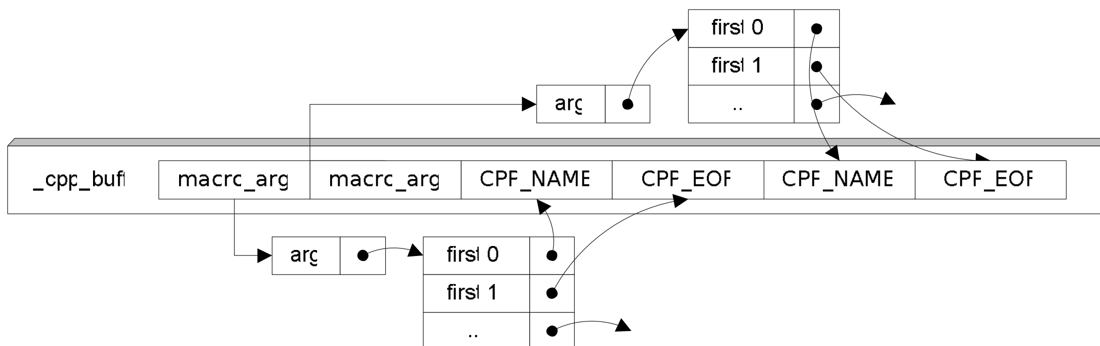


그림 1: 처리한 directive 의 적용

4. 위까지 선언된 foo 매크로의 parameter 정보를 인식하였으니, 이제 이 parameter 정보를 실제로 #define 에 선언되어 있는 것에 적용하는 것이 남았다. 즉, 교체를 해야 하는데, 그 기능을 하는 함수가 replace_args () 함수이다. 우선 실제 확장을 하기 위해 사용되는 macro_arg→expanded[...] 에 argument 들의 token 들을 넣는 것이다. 넣는 과정은 expand_arg () 함수에서 수행해 준다.

5. macro_arg→expanded[...] 에 argument 들이 들어갔다면, 이제 이 argument 들을 _cpp_buff 에 넣어야 한다. 이 _cpp_buff 에 모든 교체된 token 들이 쌓이게 되는 것이다. 물론 그 내용은 macro_arg→expanded[...] 에서 memcpy 로 복사된 것이다. 복사가 완료되면 이를 push_ptoken_context () 함수를 이용하여 새로운 context 를 만들어 등록하고 parse_in→context 에 넣는다.

parse_in→context 에 마지막으로 등록을 하고 마친 enter_macro_context () 함수부터는 이제 이 context 를 처리하는 것에 초점이 맞춰져 있다. 결과적으로 padding_token () 함수를 통해 CPP_PADDING 을 c_lex () 함수에 반환하면, 이제 context 에 기반한 처리가 이루어지기 때문이다.