Compiler Term Project

2. Parser Report

2014004211
컴퓨터소프트웨어학부
표영권

과제 요구사항

1. 이전 과제에서 구현한 C-minus lex 파일을 기반으로 컴파일
2. cminus.l과 cminus.y 파일을 yacc 프로그램으로 컴파일하여 이를 바탕으로 파싱
3. 주어진 문법[그림 1]에 맞춰서 Parse Tree가 올바르게 출력되도록 코드를 작성
4. dangling else problem의 경우, 가장 인접한 if-else가 엮이도록 코드를 작성

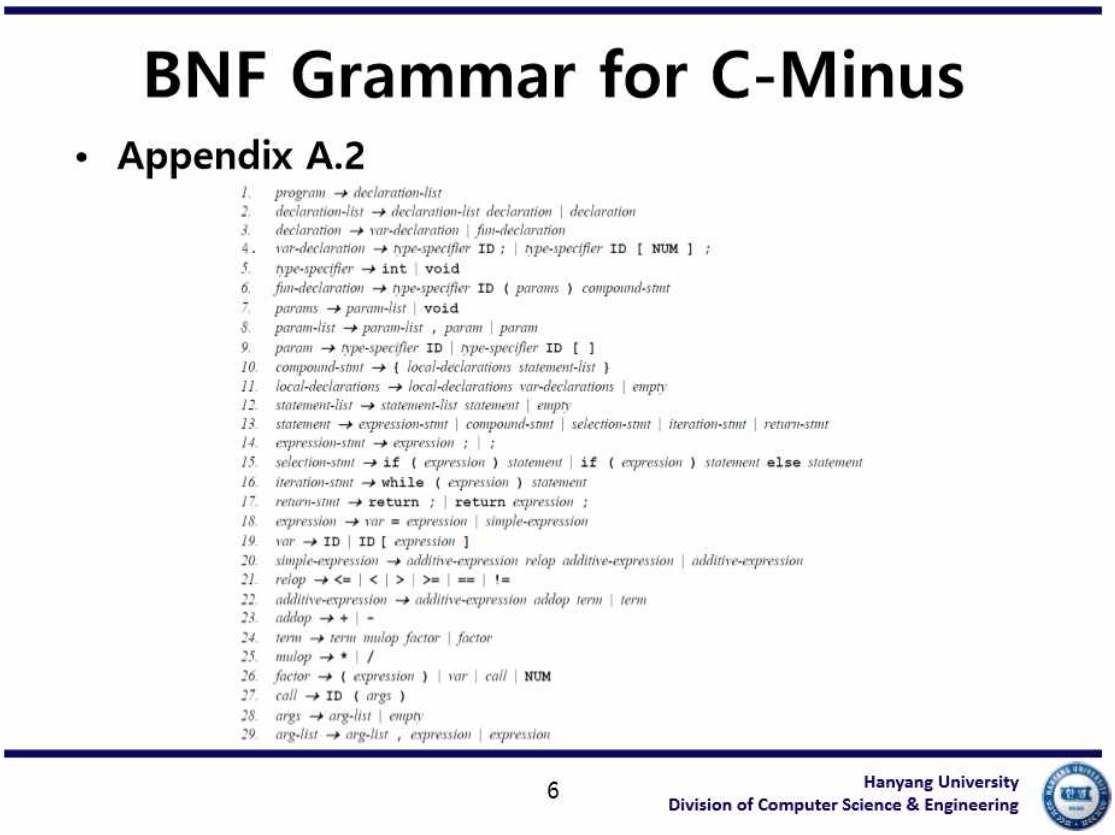


그림 1 BNF Grammar for C-Minus

실행 방법

- Build : $make
- Execution : $./cminus [test file name]
- Remove output files : $make clean

작업 환경

- OS : Ubuntu Linux 20.04 LTS
- yacc(bison), make, flex, gcc 등 컴파일하기 위해 필요한 프로그램 설치

구현 과정

Scanner project에서 완성한 파일 및 작업 환경을 이어받아 Parser project에 도입하여 시작하였다. yacc₩ 폴더에 있는 tiny.y파일을 가져와 cminus.y파일로 만들어 작업을 수행한다. 기존의 Tiny-C 문법을 모두 지우고, 주어진 C-Minus Grammar를 토대로 parsing tree를 설계하였다.

```
#define YYSTYPE TreeNode *
static char * savedName; /* for use in assignments */
static int savedLineNo;  /* ditto */
static int savedVal;
static TreeNode * savedTree; /* stores syntax tree for later return
static int yylex(void); // added 11/2/11 to ensure no conflict with
static void assert(YYSTYPE, int);

%}

%token IF ELSE RETURN WHILE INT VOID
%left PLUS MINUS
%left TIMES OVER
%token EQ NE LT LE GT GE
%token ASSIGN LPAREN RPAREN LBRACE RBRACE LCURLY RCURLY SEMI COMMA
%token ID NUM
%token ERROR

%nonassoc IFX
%nonassoc ELSE
```

그림 2 Definition part

```
65 typedef enum {PrimK,DclK,StmtK,ExpK} NodeKind;
66 typedef enum {VdclK,FdclK,TypeK} DclKind;
67 typedef enum {IfK,WhileK,ExpSK,CmpndK,RetK} StmtKind;
68 typedef enum {OpK,ConstK,IdK,AssignK,ParenK,CallK} ExpKind;
69
70 /* ExpType is used for type checking */
71 typedef enum {Void,Integer,Boolean} ExpType;
72
73 #define MAXCHILDREN 3
74
75 typedef struct treeNode
76    { struct treeNode * child[MAXCHILDREN];
77      struct treeNode * sibling;
78      int lineno;
79      NodeKind nodekind;
80      union { DclKind dcl; StmtKind stmt; ExpKind exp;} kind;
81      union { TokenType op;
82              int val;
83              char * name; } attr;
84      ExpType type; /* for type checking of exps */
85
86      int arr_size;
87      int is_param;
88    } TreeNode;
89
```

그림 3 globals.h의 Kind와 treeNode 수정

[그림 2]와 같이 Definition part를 구성하였다. 우선순위는 Bottom > Top순으로, ERROR token의 priority가 가장 높다. token의 우선순위도 중요하지만 yacc 문법을 작성하면서 Non-terminal의 decomposition 순서를 이용해 그 우선순위를 정하는 것이 가장 중요하다. instruction의 jump를 수행하거나 type match를 하는 것과 같이 빈도가 낮은 token일수록 낮은 우선순위로, 계산 순서를 즉각적으로 바꾸거나 load/store가 이루어질 수 있는 token(ASSIGN)일수록 높은 우선순위로 배치하였다.

C-Minus의 BNF의 흐름에 맞게 globals.h에 정의된 node들을 추가하였으며 C-Minus의 BNF rule에 따라 production되는 node의 종류별로 분류하였다.

treeNode 내부의 경우, 추가된 종류에 따라 union 내에 자료를 추가하였고 해당 노드가 배열로서 길이를 갖는 노드인지를 나타내기 위해 arr_size를, 어떤 함수에 종속된 parameter인지를 나타내기 위해 is_param을 추가하였다. 기본적으로 arr_size는 –1로, is_param은 FALSE로 초기화한다. 추가한 node에 맞는 node를 추가하는 함수를 util.c에 추가하였다.

```c
/* Function newPrimeNode creates a new var container
 * node for syntax tree construction
 */
TreeNode * newPrimeNode()
{ TreeNode * t = (TreeNode *) malloc(sizeof(TreeNode));
  int i;
  if(t==NULL)
    fprintf(listing,"Out of memory error at line %d\n",lineno);
  else {
    for (i=0;i<MAXCHILDREN;i++) t->child[i] = NULL;
    t->sibling = NULL;
    t->lineno = lineno;
    t->nodekind = PrimK;
    t->is_param = FALSE;
    t->arr_size = -1;
  }
  return t;
}
```

그림 4 newPrimeNode()함수. 기초적인 초기화를 다룬다.

실행 결과 ($./cminus [file name])

```c
int main(void)
{
    int i;
    i = 10;
    while(i*10 < 100) {
        i = i +1;
    }
    return 0;
}
```

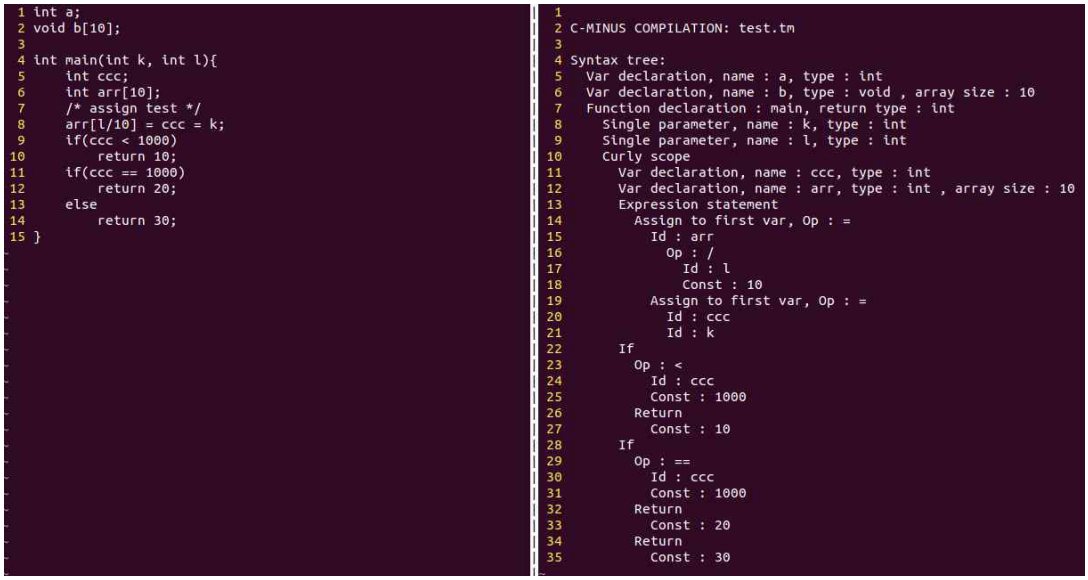
```
C-MINUS COMPILATION: while_test.tm

Syntax tree:
  Function declaration : main, return type : int
    Single parameter, name : (null), type : void
    Curly scope
      Var declaration, name : i, type : int
      Expression statement
        Assign to first var, Op : =
          Id : i
          Const : 10
      While
        Op : <
          Op : *
            Id : i
            Const : 10
          Const : 100
        Curly scope
          Expression statement
            Assign to first var, Op : =
              Id : i
              Op : +
                Id : i
                Const : 1
      Return
        Const : 0
```

그림 5 main함수 선언, local 변수 선언, assign, while의 예제

```c
int main(void) {
    int MonaWallet;
    int KeqingWallet;

    KeqingWallet = 10000;
    MonaWallet = 0;

    MonaWallet = MonaEarn(KeqingWallet/10, MonaWallet);

    return 0;
}

int MonaEarn(int sallery, int MonaWallet)
{
    return sallery + MonaWallet;
}
```

```
C-MINUS COMPILATION: Jtest.tm

Syntax tree:
  Function declaration : main, return type : int
    Single parameter, name : (null), type : void
    Curly scope
      Var declaration, name : MonaWallet, type : int
      Var declaration, name : KeqingWallet, type : int
      Expression statement
        Assign to first var, Op : =
          Id : KeqingWallet
          Const : 10000
      Expression statement
        Assign to first var, Op : =
          Id : MonaWallet
          Const : 0
      Expression statement
        Assign to first var, Op : =
          Id : MonaWallet
          Call, name : MonaEarn, with argument below
            Op : /
              Id : KeqingWallet
              Const : 10
            Id : MonaWallet
      Return
        Const : 0
  Function declaration : MonaEarn, return type : int
    Single parameter, name : sallery, type : int
    Single parameter, name : MonaWallet, type : int
    Curly scope
      Return
        Op : +
          Id : sallery
          Id : MonaWallet
```

그림 6 함수 2개 선언, 2개의 param을 받는 함수를 call하는 예제



```
int a;
void b[10];

int main(int k, int l){
    int ccc;
    int arr[10];
    /* assign test */
    arr[l/10] = ccc = k;
    if(ccc < 1000)
        return 10;
    if(ccc == 1000)
        return 20;
    else
        return 30;
}
```

```

C-MINUS COMPILATION: test.tm

Syntax tree:
  Var declaration, name : a, type : int
  Var declaration, name : b, type : void , array size : 10
  Function declaration : main, return type : int
    Single parameter, name : k, type : int
    Single parameter, name : l, type : int
    Curly scope
      Var declaration, name : ccc, type : int
      Var declaration, name : arr, type : int , array size : 10
      Expression statement
        Assign to first var, Op : =
          Id : arr
            Op : /
              Id : l
              Const : 10
          Assign to first var, Op : =
            Id : ccc
            Id : k
      If
        Op : <
          Id : ccc
          Const : 1000
        Return
          Const : 10
      If
        Op : ==
          Id : ccc
          Const : 1000
        Return
          Const : 20
        Return
          Const : 30
```

그림 7 2개 이상을 동시에 대입하는 예제 및 dangling else 예제.

구현 중 나타난 issue와 solution

순서에 맞게 parse tree가 나올 수 있도록 BNF 문법을 작성하는 것도 중요했지만, 본 과제에서 token이 입력되었을 때 해당 token의 실제 값을 읽는 코드 작성시 문제가 발생하였다. 기존에는 var_dcl -> type_spec ID SEMI 와 같은 변수 선언 BNF의 경우, type_spec을 통해 INT인지 VOID인지를 입력받고, ID에서 변수의 이름을 입력받으며 트리 노드를 구성하도록 설계하였으나 첫 Non-terminal의 reduce action이 이루어지면 뒤에 어떤 postfix가 남아있더라도 마지막 token(본 예제에서는 SEMI)을 가리키게 되는 현상이 있었다. 이를 해결하기 위하여 하나의 token들을 지날 때마다 이를 print하여 추적해본 결과, 내재적으로 yylex()를 이용해 token을 읽고 다음 token을 향하도록 설계됨을 알았다.

```
/* yylex calls getToken to make Yacc/Bison output
 * compatible with ealier versions of the TINY scanner
 */
static int yylex(void)
{ return getToken(); }
```

그림 8 yylex() 함수. 본 과제에서 직접적으로 호출하지 않는다.

여러 가지 시행착오 끝에 해당 Non-terminal의 RHS form의 마지막에서 우리는 C 코드를 작성하여 수행할 수 있고, 이 순간만큼은 token을 입력할 수 있다는 것을 깨달았으며 int a; int b;와 같이 두 개의 statement를 작성하였을 때 두 번째 statement를 아직 읽지 않았다는 사실을 통해 token을 통해 실제 값을 받는 과정은 reduce가 일어나는 순간에 작성하도록 하였고 그 결과 성공할 수 있었다. 이를 통해 [그림 11]과 같이 Non-terminal을 기존의 문법보다 더 추가하였으며 기존의 parse tree를 해치지 않도록 재설계하였다.

```
prim_dcl    : INT ID {
                /* int a */
                $$ = newPrimeNode();
                $$->attr.name = copyString(tokenString);
                $$->type = Integer;
              }
            | VOID ID {
                /* void a */
                $$ = newPrimeNode();
                $$->attr.name = copyString(tokenString);
                $$->type = Void;
              }
```

그림 9 Declaration. prim_dcl의 reduce를 통해 ID를 얻을 수 있도록 하였다.

해당 이슈를 해결한 부분은 prim_dcl이다. newPrimeNode() 함수는 TreeNode를 생성하되 가장 기본적인 값으로 초기화하여 Node를 생성하는 함수이다.

```
select_stmt : IF LPAREN exp RPAREN stmt
                { $$ = newStmtNode(IfK);
                  $$->child[0] = $3;
                  $$->child[1] = $5;
                } %prec IFX
            | IF LPAREN exp RPAREN stmt ELSE stmt
                { $$ = newStmtNode(IfK);
                  $$->child[0] = $3;
                  $$->child[1] = $5;
                  $$->child[2] = $7;
                }
```

그림 10 dangling else의 해결

본 과제에서 주어진 dangling else의 해결에 대해서는 priority를 지정하여 else가 나타나는 경우 가장 가까운 if에 매칭되도록 문법을 지정할 수 있었다([그림 12] 참조).

If-Else Ambiguity

言語を選択 Powered by Google 翻訳

A shift-reduce conflict that frequently occurs involves the *if-else* construct. Assume we have the following rules:

```
stmt:
   IF expr stmt
   | IF expr stmt ELSE stmt
   ...
```

and the following state:

```
IF expr stmt IF expr stmt . ELSE stmt
```

We need to decide if we should shift the **ELSE** or reduce the **IF expr stmt** at the top of the stack. If we shift then we have

```
IF expr stmt IF expr stmt . ELSE stmt
IF expr stmt IF expr stmt ELSE . stmt
IF expr stmt IF expr stmt ELSE stmt .
IF expr stmt stmt .
```

where the second **ELSE** is paired with the second **IF**. If we reduce we have

```
IF expr stmt IF expr stmt . ELSE stmt
IF expr stmt stmt . ELSE stmt
IF expr stmt . ELSE stmt
IF expr stmt ELSE . stmt
IF expr stmt ELSE stmt .
```

where the second **ELSE** is paired with the first **IF**. Modern programming languages pair an **ELSE** with the most recent unpaired **IF**. Consequently the former behavior is expected. This works well with yacc because the default behavior, when a shift-reduce conflict is encountered, is to shift.

Although yacc does the right thing it also issues a shift-reduce warning message. To remove the message give **IF-ELSE** a higher precedence than the simple **IF** statement:

```
%nonassoc IFX
%nonassoc ELSE

   stmt:
       IF expr stmt %prec IFX
       | IF expr stmt ELSE stmt
```

그림 11 If-Else Ambiguity