hw3b

B511209 최아성

May 2019

1 Lex-Yacc설계 설명

1.1 Lex

Lex는 먼저 파일의 코드들을 의미 있는 토큰들로 토큰화 시키는데 목적이 있다.

크게 4가지 기준으로 토큰화 하였다.

1. 식별자 IDENTIFIER : '_'와 영문자중 하나로 시작하며,
숫자도 포함할 수 있다.
2. 연산자, c언어에서 사용하는 특수문자 연산자
3. 예약어
4. 숫자 16진수(0x로 시작)와 8(0으로 시작)진수도 가능하다. 그외 문자열도 있다.

1.2 Yacc

Yacc는 코드의 문법을 정의하는데 목적이 있다.

크게 독립적으로 분류 가능한 7가지로 분류하였다.

1. 함수 정의 : 함수 안의 실행문 내용을 정의한다.
2. 변수(함수) 선언 : 변수들을 선언한다. 함수 선언도 여기 포함된다.
3. 반복문 : for , do while , while문이 해당한다.
4. 선택문 : if , switch문이 해당한다.
5. 분기문 : goto , return, break 등이 해당한다.
6. label: case: , default: , 식별자 :, 등이 해당한다.
7. 수식 : 변수들의 연산, 혹은 함수 호출 등이 해당한다.
그리고 이 7가지의 실행문들을 중괄호로 담을 수 있는 compound : '실행문들의 반복'가 있다

이 7+1가지를 합쳐 코드에서는 statement로 정의 하였다.
이 중 가장 처음 전역공간에서 사용이 가능한것은 1,2번 뿐이다.
나머지는 statement를 사용하는 문법들에서 사용가능하다. ex) 함수의 정의부

2 Lex 코드및 설명

```
%{
#include<stdio.h>
#include"y.tab.h"
%}

D [0-9]
L [a-zA-Z]
A [a-zA-Z_0-9]
P {D}+\.{D}+      //실수 표현

 //예약어

%%
"include" {return INCLUDE;}
"define" {return DEFINE;}
"int" {return INT;}
"char" {return CHAR;}
"float" {return FLOAT;}
"void" {return VOID;}
"for" {return FOR;}
"if" {return IF;}
"while" {return WHILE;}
"do" {return DO;}
"return" {return RETURN;}
"auto" {return AUTO;}
"break" {return(BREAK); }
"case" {return(CASE); }
"const" {return(CONST); }
"continue" {return(CONTINUE); }
"default" {return(DEFAULT);}
"double" {return(DOUBLE); }
"else" { return(ELSE); }
"enum" { return(ENUM); }
"extern" { return(EXTERN); }
"goto" { return(GOTO); }
"long" { return(LONG); }
"register" { return(REGISTER); }
"short" { return(SHORT); }
"signed" { return(SIGNED); }
"static" { return(STATIC); }
"struct" { return STRUCT; }
"switch" { return SWITCH; }
"typedef" { return TYPEDEF; }
```

```
"union"  { return UNION; }
"unsigned" { return UNSIGNED; }
"volatile" { return VOLATILE;}
L?\"(\\.|[^\\"])*\" {return LITERAL;}  //문자열
 //피연산잔 두개를 가지는 operator
"->" {return POINTER;}
"&&" {return AND;}
"||" {return OR;}
"==" {return EQUAL;}
"sizeof" {return SIZE;}
"!=" {return DIFF;}
"<=" {return RREL;}
">=" {return LREL;}
"+=" {return ADDE;}
"-=" {return SUBE;}
"/=" {return DIVE;}
"*=" {return MULE;}
"%=" {return QUEE;}
">>=" {return RIGHTE;}
"<<=" {return LEFTE;}
"^=" {return XORE; }
"|=" {return ORE;}
"&=" {return ANDE;}
"<<" {return LSHF;}
">>" {return RSHF;}
 //숫자 NUMBER
 //10진수 와 8진수
{D}+ |
 {P}+   {return NUMBER;}
  //16진수
0(X|x)({D}+|{P}+)  {return NUMBER;}
"++" {return PP;}
"--" {return MM;}
 //피연산잔 한 개를 가지는 operator
[;,\[\]&{}=*()\-\\#^!><|/%~+.~:?] {return yytext[0];}
 식별자 IDENTIFIER
{L}{A}* {return IDENTIFIER;}
[\t\v\f\n] {}
.       {}
%%

int yywrap(){
        return 1;
}
```

3 Yacc 코드와 설명

c언어의 문법 체계에 맞췄음으로 c++은 구문 오류가 날 수 있습니다.

3.1 흐름의 따른 문법 분석

위에서 언급한 7가지 분류\\\\

1. 수식 : expression

id : 가장 기본, 문자(열), 식별자, 숫자, '(' ')' 둘러 쌓인 수식
tValue: 후위 연산, 함수, 배열
sValue: 멤버 접근자 (c는 구조체에 함수가 없다. tValue안에 포함해도 된다)
uValue: 전위 연산자 , sizeof(type_name)
oValue: 타입 캐스팅, (type_name) oValue (전위와 겹쳐서 reduce오류 예방을 위해 나눔)
mValue: 두가지 피연산자를 갖는 연산들(문법상에 맞춰서 우선순위 고려하지 않음)
cValue: ? ~ : ~ 연산자
Value: 배정문을 포함한 수식표현

2. 변수(함수) 선언: declaration

direct_declaratior : 식별자 변수, 함수, 배열의 모양
declarator: 식별자에 포인터의 여부
init_declarator : 식별자 정의에 배정식의 여부
init_declarator_list: 위에서까지 정리된 식별자 문법은 ','을 통해 반복가능
declaration: 타입만, or 타입 + 식별자 선언형태

3. 함수 정의: function
타입 declarator 정의부(conpount_statement)

4. 선택문 : section_statement
if문 else문 swith문 , 코드참조

5. 반복문 : iteration_statement
for , while, do while, 코드참조

6. label : label_statement
식별자: 라벨, case : 라벨 , default : 라벨

7. 분기문 : jump_statment
goto문 continue문 break문 return문

3.2 카운팅 방식

1.수식

대부분 expression의 하위 문법에서 reduce하며 카운트
(코드 참조, init_declarator에서 배정문도 카운트)

2.함수

함수 호출은 tValue에서 함수형태의 모양이 reduce시 카운트
그외에는 direct_declaration에서 함수형태 모양으로
reduce시 카운트,함수체크를 해준다
함수를 체크 하는 이유는 상위 reduce에서 포인터나 변수로서 카운팅 되지 않기 위해.

3. 배열 선언

direct_declaration에서 배열의 모양으로 reduce되었다면
배열체크를 해준다.
배열모양마다 카운팅할경우 생기는 다차원 배열의 다수 카운팅 방지
declarator에서 reduce시 배열체킹이 되었다면 배열을 카운팅해준다.

4. 포인터 선언

declarator에서 포인터 형태의 모양으로 reduce시 카운트해준다.

5. 변수 선언

declarator에서 함수체크가 안되있다면 항상 T를 더해준다
declaration에서 타입에 따라 결정된 type변수의 상태에 따라
알맞은 타입에 T만큼 카운트를 더해준다.
그 후 초기화 되야 할 값들은 초기화 시켜줌.

6. 반복, 선택 기타등등

return문은
분기문 문법안에
return; 과 reutrn cValue ; reduce시 카운팅
(문자열의 retrun의 형태는 토큰화할 때 구분됨)
나머지는 항상 statement에서 reduce가 일어나기에
statement의 알맞는 reduce형태에 따라 카운팅한다

```
%{
#include<stdio.h>
int yylex();
int count[3] ={0};   2는 int, 1은 char, 0은 나머지
int T =0;            변수 선언의 개수 저장
int P = 0;           포인터 선언 개수 저장
int A = 0;           배열 선언 개수 저장
int checkArr = 0;    배열 선언인지 검사
int checkFunct = 0;  함수 선언인지 검사
int funct = 0;       함수 선언,호출 개수 저장
int op=0;            연산자 개수 저장
int selection =0;
int iteration =0;
int return_ =0;
int type = 0;        count에 저장할 타입을 결정
%}

%token INCLUDE
%token DEFINE
%token NUMBER IDENTIFIER
%token INT VOID
%token CHAR
%token FLOAT
%token PP            *"++"
%token MM            "-"
%token POINTER       "-¿"
%token AND
%token OR
%token DIFF          "!="
%token SIZE          "sizeof"
%token EQUAL         "=="
%token ADDE          연산 배정문 ex "+="
%token MULE
%token DIVE
%token QUEE          "%="
%token SUBE
%token RREL
%token LREL
%token LSHF
%token RSHF
%token LEFTE
%token RIGHTE
%token XORE          "≐"
%token ORE
%token ELSE
```

```
%token ANDE
%token LITERAL        문자열 상수
%token CASE DEFAULT IF ELSE SWITCH WHILE DO FOR GOTO CONTINUE BREAK RETURN
%token TYPEDEF EXTERN STATIC AUTO REGISTER
%token SHORT LONG SIGNED UNSIGNED DOUBLE CONST VOLATILE
%token STRUCT UNION ENUM
%start start_state
%%

 실제 시작은 sub_start를 좌결합으로 반복
start_state :sub_start
        | start_state sub_start  ;


 변수나, 숫자, 함수등을 호출, 사용 하는 모든 수식 표현들
 수식들의 ','을 통한 좌결합 연결 : 함수 호출 시 인자로 사용가능
expression : Value
        | expression ',' Value
        ;

 배정문 표현
 우결합 배정문으로, 배정문을 우측으로 반복 가능하다
Value : cValue
        | oValue assign_oper Value {op++;}
        ;

 ? : 논리비교 연산자
 ? : 우결합
cValue : mValue
        | mValue '?' expression ':' cValue {수정 제거 op++;}
        ;


 피연산자 2개 연산 표현
 리프에 가까울 수록, 결합방향(좌,우)의 끝일 수록 연산 우선순위가 높지만
 문법확인과 카운팅이 목적이기 원래 c언어의 연산자 우선순위는 고려하지 않았다
mValue : oValue
        | mValue '*' oValue     {op++;}
        | mValue '/' oValue     {op++;}
        | mValue '%' oValue     {op++;}
        | mValue '+' oValue     {op++;}
        | mValue '-' oValue     {op++;}
        | mValue '>' oValue     {op++;}
        | mValue '<' oValue     {op++;}
        | mValue  LREL  oValue {op++;}
        | mValue  RREL  oValue {op++;}
```

```
        | mValue RSHF oValue {op++;}
        | mValue LSHF oValue {op++;}
        | mValue '&' oValue    {op++;}
        | mValue  AND oValue   {op++;}
        | mValue '|' oValue    {op++;}
        | mValue OR oValue     {op++;}
        | mValue EQUAL oValue  {op++;}
        | mValue '^' oValue    {op++;}
        | mValue  DIFF  oValue {op++;}

 전위 연산자와 type_name이 겹쳐 reduce에러 방지를 위해
 type 캐스팅!
oValue : uValue
        | '(' type_name ')' oValue
        ;

 전위 연산자 우결합
uValue : sValue
        | MM uValue {op++;}
        | PP uValue {op++;}
        | SIZE uValue
        | SIZE '(' type_name ')'
        | unray_oper uValue
        ;

 멤버접근자로 함수나 배열의 인자도 부를 수 있다.
 tValue는 좌결합 문법인데, 우측에 함수호출 문법인 tValue를 쓰면 오류발생,
 그래서 멤버 접근자 문법 sValue를 좌결합형태로 만들었다. ,

 수정 ..../c++은 가능하나, C에서 구조체 안에 함수는 사용 불가하다
 tValue가 아닌 IDENTIFIER가 더 정확
 (tValue가 IDENTIFIER를 포함하여 오류는 없다.)
sValue : tValue
        | sValue '.' tValue
        | sValue POINTER tValue  {op++;}
        ;

 후위 연산자 좌결합
tValue : id
        | tValue '(')' {funct++;}
        //수식에서 배열은 인덱스 접근만 가능하다.
        | tValue '['expression']'
        | tValue '(' expression ')'{ funct++;}
        | tValue PP  {op++;}
        | tValue MM {op++;}
        ;
```

```
 c언어에서 tpye의 종류
 type 뒤에 const, '*' 등이 올 수 있다. 좌결합
type_name: typeMix
         |  typeMix pointer
         ;

 type 앞에 const등이 올 수 있다. 우결합
typeMix: type_specifier
         | type_qualifier typeMix
         ;

 모든 배정연산자
assign_oper: '='
         | ADDE
         | MULE
         | DIVE
         | QUEE
         | SUBE
         | LEFTE
         | RIGHTE
         | ORE
         | XORE
         | ANDE
         ;


단일 피연산자 연산자
unray_oper: '&'
         | '*'
         | '+'
         | '-'
         | '~'
         | '!'
         ;

변수(함수)선언시 타입 형태 종류
type_specifier: CHAR     {type = 1;} char형은 인덱스 '1'
         | INT           {type = 2;} int형은 인덱스 '2'
         | FLOAT
         | VOID                      기본 타입들
         | DOUBLE
         | SHORT
         | LONG
```

```
        | SIGNED
        | UNSIGNED
        | struct_or_union_specifier  struct형 선언, 정의
        | enum_specifier   열겨형 선언, 정의
        ;

 열겨형의 문법 형태 3 가지
enum_specifier
        : ENUM '{' enumerator_list '}'
        | ENUM IDENTIFIER '{' enumerator_list '}'
        | ENUM IDENTIFIER
        ;

 열거형 정의부 내부의 들어갈 내용들
enumerator_list
        : enumerator
        | enumerator_list ',' enumerator
        ;
enumerator
        : IDENTIFIER
        | IDENTIFIER '=' cValue
        ;


 const 와 valatile
type_qualifier
        : CONST
        | VOLATILE
        ;
type_qualifier_list
        : type_qualifier
        |type_qualifier_list type_qualifier\
        ;

 타입과 변수(함수) 선언
declaration : declaration_specifiers
        | declaration_specifiers init_declarator_list
        {count[type] +=T; type = 0; T = 0;}
        ;

        변수 선언시 타입을 제외한 나머지 부분

 식별자는 ','로 구분하여 한번에 여러개 선언이 가능하다
init_declarator_list
        : init_declarator
        | init_declarator_list ',' init_declarator
```

```
        ;

 식별자는 선언과 동시에 값을 배정 할 수도 있고 안할 수도 있다.
init_declarator
        : declarator
        | declarator '=' initializer {op++;}
        | declarator ':' cValue {op++;} 구조체 내부 선언시 문법
        | ':' cVlalue {op++;}
        ;

 식별자가 포인터인지 처리
 함수 선언이라면, 변수 선언 개수 값인 T와 포인터 선언 수인 P를 더하지 않는다
declarator
    : direct_declarator
{if(!checkFunct){T++;} if(checkArr){A++; checkArr = 0;} checkFunct = 0; }
        | pointer direct_declarator
{if(checkArr){A++; checkArr = 0;} if(!checkFunct) {P++;T++;} checkFunct = 0;}
        | pointer   함수 선언시 매개변수로 식별자 없이 *만 (int *,char*)
        ;

 변수 선언에서 포인터로서 선언될 수 있는 문법들
pointer: '*'
        | '*' type_qualifier_list
        | '*' pointer
        | '*' type_qualifier_list pointer
        ;

 변수의 기본형들 (일반 식별자,함수,배열)
direct_declarator: IDENTIFIER
    | '(' declarator ')' 함수인지 check한다
    | direct_declarator '(')'{funct++; checkFunct = 1;}
    | direct_declarator'[']' {checkArr = 1;}
                            배열인지 체크 다차원 배열 중복방지
    | direct_declarator'[' cValue ']'              {checkArr = 1;}
                            cValue는 상수연산을 포함하고있음
    | direct_declarator'('identifier_list')' {funct++; checkFunct = 1;}
    | direct_declarator'(' parameter_list')' {funct++; checkFunct = 1;}
        ;


 함수 정의시 필요한 인자형태
parameter_list
        : parameter_declaration
        | parameter_list ',' parameter_declaration
        ;
 C는 디폴트 매개변수가 없어서 declaration과 다르다.
```

```
parameter_declaration
        : declaration_specifiers declarator {count[type] +=T; type = 0; T = 0;}
        | declaration_specifiers함수선언시 매개변수는 타입만 선언
        ;


identifier_list
        : IDENTIFIER
        | identifier_list ',' IDENTIFIER
        ;


 선언시 배정 할 수 있는 값의 형태
initializer
        :  Value
        | '{' initializer_list '}'
        | '{' initializer_list ',' '}'
        ;
initializer_list
        : initializer
        | initializer_list ',' initializer
        ;



 선언시 타입의 종류와 추가기능(static,const등등)..포함
declaration_specifiers
    : storage_class_specifier
        | storage_class_specifier declaration_specifiers
        | type_specifier
        | type_specifier declaration_specifiers
        | type_qualifier
        | type_qualifier declaration_specifiers
        ;


 구조체, union
struct_or_union :UNION
        |STRUCT
        ;

 구조체 선언 or 정의
struct_or_union_specifier: struct_or_union IDENTIFIER
        | struct_or_union IDENTIFIER '{''}'
        | struct_or_union IDENTIFIER '{'struct_union_parameter_list '}'
        ;
```

```
 구조체의 정의부
struct_union_parameter_list : struct_union_parameter
        | struct_union_parameter_list  struct_union_parameter

 구조체 정의부는 변수와 함수 선언,정의만 가능
struct_union_parameter: declaration ';' | function;
수정 //c++과 달리 C는 구조체 안에서 함수 선언이 불가하다.
수정 //funtion은 무시해도 좋습니다.

  expression의 가장 기본
id : IDENTIFIER
        | NUMBER
        | '(' expression ')'
        | LITERAL
        ;

 변수 선언시 추가적인 기능을 사용해주는 예약어
storage_class_specifier : TYPEDEF
        | EXTERN
        | STATIC
        | AUTO
        | REGISTER
        ;
 실행 할 수 있는 추상화된 명령어들의 모임
statement : declaration ';' 변수,함수 선언
    | expression_statement  연산,수식 기본 표현들
    | function              함수 선언과 정의
    | compound_statement    명령어들의 나열을 포함하고 있는 실행문
    | selection_statement   {selection++;}선택문
    | iteration_statement   {iteration++;}반복문
    | labeled_statement                   레이블
    | jump_statement                      분기문
        ;

 statement들의 모임 좌결합
statement_list: statement
        | statement_list statement
        ;

 "{}" 중괄호 안에 실행되는 명령어들을 담는다
compound_statement
        :'{' statement_list '}'
        |'{' '}'
        ;
```

statement의 종류들

```
expression_statement
        : ';'
        | expression ';'
        ;

jump_statement
        : GOTO IDENTIFIER ';'
        | CONTINUE ';'
        | BREAK ';'
        | RETURN ';' {return_++; }
    return 문은 함수안에서만 실행되며 함수 종료와 반환값을 주는 역할
    함수 밖(전역)에서 사용가능한 명령어는 함수와 변수 선언 뿐이다.
        | RETURN expression ';'{return_++; }
        ;


labeled_statement
        : IDENTIFIER ':' statement
        | CASE cValue ':' statement
        | DEFAULT ':' statement
        ;


selection_statement
        : IF '(' expression ')' statement
        | ELSE statement
        else문도 읽을 수 있지만 if문 없이 단독 사용이 가능해서 불완전함
        | SWITCH '(' expression ')' statement
        ;

iteration_statement
        : WHILE '(' expression ')' statement
        | DO statement WHILE '(' expression ')' ';'
        | FOR '(' expression_statement expression_statement ')' statement
| FOR '(' expression_statement expression_statement expression ')' statement;



 함수 정의
 함수개수의 합은 declarator에서 더한다.
 안정성을 위해 리셋이 필요한 값들만 건들여준다.
function : declaration_specifiers declarator compound_statement
    {type=0;T=0;checkFunct = 0;}
        ;
```

```
 include 헤더의 일반적인 형태만 넣음
include : '#' INCLUDE '<' header '>'
        | '#' INCLUDE LITERAL
        | '#' INCLUDE '<' IDENTIFIER'>'
        ;
header: IDENTIFIER '.' IDENTIFIER
        ;


 전처리기는 정의 부분에 제한이 없고
 구분이 줄바꿈이어서 reduce가 쉽지않아
 자주사용하는 한정된 구문만 정의했습니다.
def     : '#' DEFINE IDENTIFIER IDENTIFIER
        | '#' DEFINE IDENTIFIER NUMBER
        ;

 전역 부분에 사용 가능한 문법
 처음 전역 부분에서는 선언과 정의만 가능하다.
sub_start:  declaration ';'
        | function  //실행문들은 함수의 정의부분 안에서 가능
        |';'        //문장 어디서든 구분자는 중복 사용 가능하다
        | include
        | def
        ;

%%

void yyerror(const char *str)
{
        fprintf(stderr,"error : %s\n",str);
}
int main(void){

        printf("\n변수 선언은 함수의 선언부분의 매개변수도 포함합니다.\n\n");
        yyparse();
        printf("int 변수 선언 = %d\n",count[2]);
        printf("char 변수 선언 = %d\n",count[1]);
        printf("포인터 변수 선언 = %d\n",P);
        printf("배열 변수 선언 = %d\n",A);
        printf("함수 (호출,선언,정의) =  %d\n",funct);
        printf("수식 =  %d\n",op);
        printf("반복문 = %d\n",iteration);
        printf("선택문 = %d\n",selection);
        printf("return문 = %d\n",return_);
```

```
        return 0;
}
```

4 후기

실제 yacc를 설계하면서 c언어에 어떤 문법이 실제로 돌아가는지 많이 테스트 해 보았다.

배열의 인덱스를 수식으로 사용할 때 인덱스 값에 expression이 들어가는데

정말 문법에 arr[1,2,3]과 같은 형태가 오류 없이 사용 가능했다.

(물론 arr[3]과 동일하여 의미가 더 있는 형태는 아니 였다.)

그 외에도 statement를 사용하는 문법들을 보며 for문 if문 등등이

중괄호로 둘러 쌓인 실행문 외에도 한개의 또다른 statement를 어떤 원리로 사용하는지 이해할 수 있었다.

그리고 수식과 실제 c에서는 마치 (int *)는 (int)는 또다른 변수 타입처럼 사용되는데,
(타입 캐스팅, 컴파일시 int형 변수와 int * 변수는 차별을 둔다.)

변수를 선언할 때 포인터는 int, char, float 과 같이 (int *,char *)는 따로 구분되지 않는다.
(int *a,b,c; int *형으로 모두 선언이 아니다, a만 포인터 선언이고 나머지는 int 형이다.)

선언시 포인터의 구분은 식별자명의 앞에 ' * '로 구분짓는다.

이 원리도 문법을 정의하면서 선언이 수식의 연장선이 아닌
약간은 새롭게 정의 되었다는 것을 알 수 있었다.

언어의 형태나 기능을 조금 더 심도있게 바라본거 같았다.–