컴파일러 과제-5

20192800 권대현

1. 과제 내용

* 이번 컴파일러 과제는 2장의 문법(lex)과 3장에서 설명한 어휘 분석 프로그램(yacc), 4장, 5장에서 설명한 내용을 토대로 C언어의 신택스 분석기를 제작하여 C언어 코드의 문법을 검사하는 것이다.
* 입력으로 선언문, 명령문 및 함수가 포함된 프로그램들이 주어진다.
* 수식이 잘못된 경우 line 번호와 함께 syntax 오류가 어디서 일어났는지를 출력한다.
* 수식이 올바를 경우 syntax tree를 출력하여 신택스 분석이 된 과정을 보여준다.

1. 문제 및 해결 방법

* C언어 파서의 문법은 컴파일러-2장 강의노트.pdf 전체를, yacc 프로그램과 lex 문법은 컴파일러-3장 강의노트.pdf를 참고하여 작성했다.
* Lex와 yacc 명령어를 터미널에서 사용하기 위해 bison과 flex 패키지를 설치했다.
* 처음 yacc 프로그램을 돌렸을 때 중괄호{ }나 , 콤마를 잘못 사용하여 오류가 났었다. 이를 각각 LR RR, COMMA로 토큰명으로 바꿔줬다.
* 2장의 강의노트 pdf를 통해 yacc 문법을 작성할 때 콜론 : 문양을 잘못 이해하여 에러가 많이 떴었다. 이를 COLON으로 토큰명으로 고쳐서 해결했다.
* 이번 과제에서 type\_identifier를 구분 짓는 함수를 제외해야 했기 때문에 따로 int, float, void, char 형의 토큰을 추가하였다.
* 2개의 Shift/Reduce conflicts가 발생했다. 하나는 사전에 알고 있던 if – else ambiguity 문제이기에 넘어갔다.
* 다른 하나는 unary\_expression = assignment\_expression (SHIFT)와 unary\_expression -> constant\_expression = initializer (REDUCE)에 대한 상호 충돌이었다. 이 문제는 콤마 기호의 혼란을 없애기 위해 constant\_expression 쪽 문법과 initializer의 문법에서 expression을 assignment\_expression으로 변경해서 생긴 문제이다. 해당 conflicts를 해결하여 shift/reduce 개수를 줄여도 parser에서 제대로 parsing을 하지 않는 문제가 발생하였기에 그대로 두었다.
* 나머지는 reduce/reduce conflicts가 있다. Reduce/reduce의 경우 해결하기 위해 새로 예외 룰을 추가하거나 토큰 분류를 보다 상세히 해야 한다. 그러나 parser에서 문법의 오류를 검출하기엔 문제가 없음을 확인하여 그대로 두었다.
* 3장의 강의노트 pdf를 통해 lex 파일을 만들던 중 /\* \*/의 주석을 처리하는 정규식에 오류가 있음을 확인했다. 따라서 동일한 기능을 수행하는 정규식으로 코드를 수정했다.
* 자르는 토큰에 대해서 yylval에 어떻게 값을 넘겨줄지 고민이 됐다. 이내 yytext를 넘겨주는 방식이 생각이 났고, 적절한 형변환을 통해 정수값, 문자값, 스트링 주소로 yytext를 넘겨줬다.

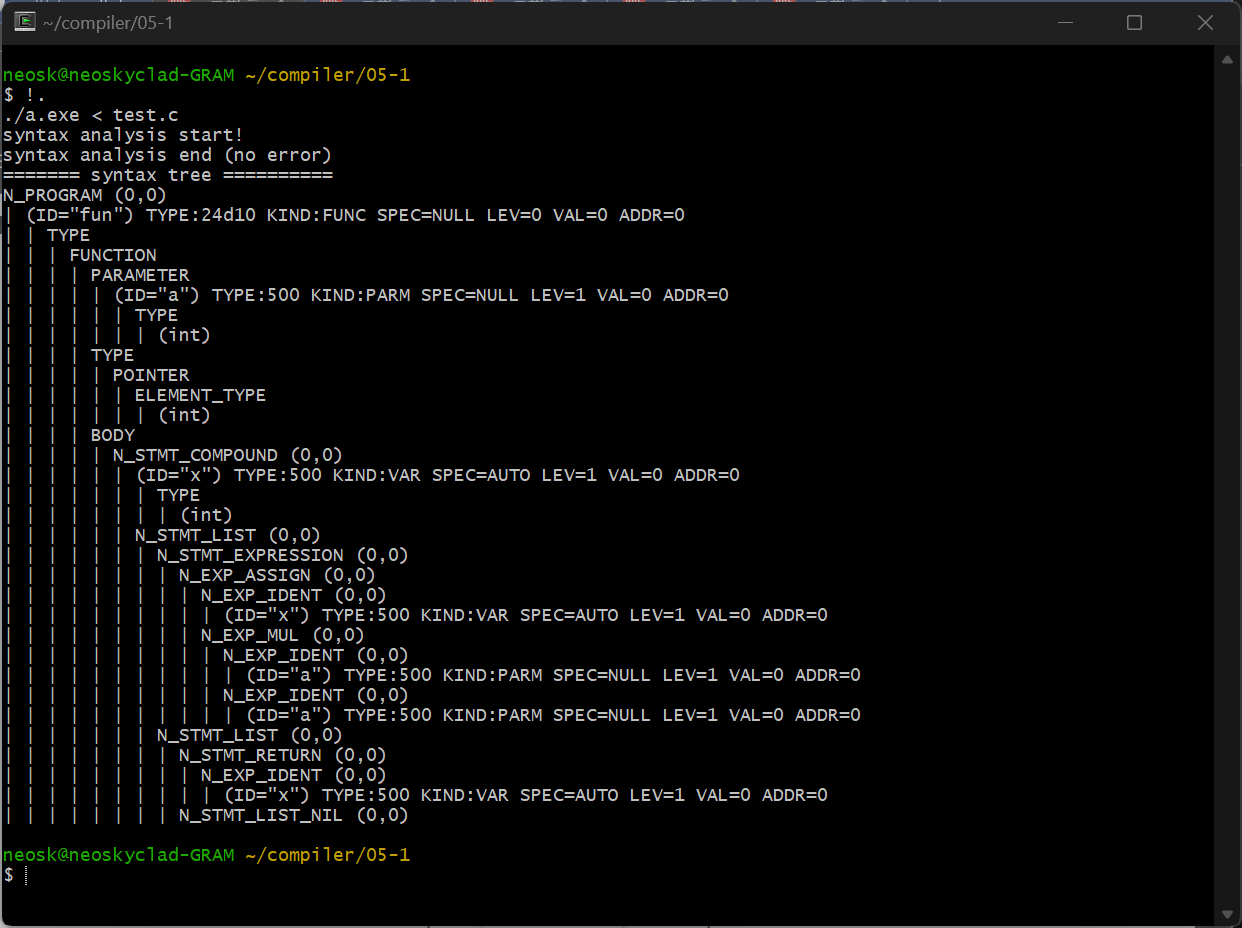
1. 테스트

* 올바른 C언어 코드
  + Hello World 출력

텍스트, 스크린샷, 소프트웨어, 멀티미디어 소프트웨어이(가) 표시된 사진

자동 생성된 설명

* + - Main 함수에서 printf(“Hello World!\n”)를 수행했다.
  + Int \*fun()함수



* + - 파라미터로 int a를 넘겨받고 지역변수로 int x를 선언하는 int \*fun()함수를 선언한다.
    - X = a\*a;와 return x를 반환한다.
  + Enum typedef와 struct 지역변수 테스트

텍스트, 전자제품, 스크린샷, 컴퓨터이(가) 표시된 사진

자동 생성된 설명

* + - Typedef enum {false, true} BOOLEAN으로 bool형 자료형을 정의한다.
    - BOOLEAN형으로 fun() 함수를 선언했다.
      * Fun()함수 내부에는 static int x; auto struct { int a; char c; } s; float I, j, k가 존재한다.
      * 이것으로 int형, struct형, float형, char형의 선언이 동작하는 것을 확인할 수 있다.
  + Struct 구조체 선언 테스트

텍스트, 전자제품, 스크린샷, 컴퓨터이(가) 표시된 사진

자동 생성된 설명

* + - Struct의 일반적인 선언, struct의 불완전선언을 테스트해봤다.
  + Enum 선언 테스트

텍스트, 전자제품, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

* + - Enum 형으로 color을 정의하고, color 내부 멤버를 초기화하는 코드를 선언했다.
    - Enum color c1, c2; 를 통해 사전 정의된 color형을 사용하는 변수 선언을 테스트 했다.
  + 다양한 declarator 형태 테스트

텍스트, 스크린샷, 소프트웨어, 디스플레이이(가) 표시된 사진

자동 생성된 설명

* + - Int \*p, float \*f(), float (\*b[10])(int)와 같이 다양한 형태의 declarator를 테스트 했다.
  + If문 테스트

텍스트, 전자제품, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

* + - If(expression) if(expression) else statement 형태의 if문을 테스트해봤다.
    - If-else ambiguity가 있는 문장이지만, 신택스 분석에는 문제가 없다.
  + 반복문 테스트

텍스트, 전자제품, 스크린샷, 디스플레이이(가) 표시된 사진

자동 생성된 설명

* + - while문과 for문을 테스트 해봤다.
  + Switch문과 jump문 테스트

텍스트, 전자제품, 스크린샷, 컴퓨터이(가) 표시된 사진

자동 생성된 설명

* + - switch문은 사전에 정의된 i를 기준으로 검사된다.
      * Case 0: break;를 통해 case문과 break문의 동작을 테스트했다.
      * Case 1: continue;를 통해 continue문을 테스트했다.
      * Default: return;를 통해 default label과 return문을 테스트했다.
  + Postfix expression

텍스트, 전자제품, 스크린샷, 디스플레이이(가) 표시된 사진

자동 생성된 설명

* + - 다양한 postfix 수식을 테스트했다.
      * Int a[10];
      * Int b();
      * Six.a++;
      * Six->b--;
* 잘못된 C언어 코드
  + 잘못된 함수 선언

텍스트, 전자제품, 스크린샷, 디스플레이이(가) 표시된 사진

자동 생성된 설명

* + - Main 함수를 선언할 때 닫는 중괄호를 생략해봤다.
    - Line 2에서 에러가 발생했다.
  + 선언되지 않은 변수 접근

텍스트, 소프트웨어, 멀티미디어 소프트웨어, 폰트이(가) 표시된 사진

자동 생성된 설명

* + - 선언되지 않은 a, b를 코드에서 사용해봤다.
  + 파라미터 관련 오류

텍스트, 소프트웨어, 멀티미디어 소프트웨어, 그래픽 소프트웨어이(가) 표시된 사진

자동 생성된 설명

* + - 먼저 a를 정의한 뒤, 뒤에서 한 번 더 정의하여 redeclaration 오류를 나타냈다.
    - Int fun(int)라는 프로토타입을 선언하고 함수를 구현할 때 파라미터를 char형으로 달리하여 conflicting parm type 오류를 나타냈다.
    - Int fun2(int)라는 프로토타입을 선언하고 함수를 구현할 때 int fun2(int)로 파라미터의 이름을 넣지 않아서 empty parameter name 오류를 나타냈다.

1. 원시프로그램

* Lex

digit [0-9]

letter [a-zA-Z\_]

delim [ \t]

line [\n]

ws {delim}+

%{

#define YYSTYPE\_IS\_DECLARED 1

typedef long YYSTYPE;

#include "y.tab.h"

#include "type.h"

#include <stdlib.h>

#include <string.h>

char \*makeString(char \*);

int checkIdentifier(char \*);

%}

%%

{ws} { }

{line} { }

auto { return (AUTO\_SYM); }

break { return (BREAK\_SYM); }

case { return (CASE\_SYM); }

continue { return (CONTINUE\_SYM); }

default { return (DEFAULT\_SYM); }

do { return (DO\_SYM); }

else { return (ELSE\_SYM); }

enum { return (ENUM\_SYM); }

for { return (FOR\_SYM); }

if { return (IF\_SYM); }

return { return (RETURN\_SYM); }

sizeof { return (SIZEOF\_SYM); }

static { return (STATIC\_SYM); }

struct { return (STRUCT\_SYM); }

switch { return (SWITCH\_SYM); }

typedef { return (TYPEDEF\_SYM); }

union { return (UNION\_SYM); }

while { return (WHILE\_SYM); }

goto { return (GOTO\_SYM); }

"\+\+" { return (PLUSPLUS); }

"\-\-" { return (MINUSMINUS); }

"\->" { return (ARROW); }

"<" { return (LSS); }

">" { return (GTR); }

"<=" { return (LEQ); }

">=" { return (GEQ); }

"==" { return (EQL); }

"!=" { return (NEQ); }

"&&" { return (AMPAMP); }

"||" { return (BARBAR); }

"<<" { return (LSH); }

">>" { return (RSH); }

"\.\.\." { return (DOTDOTDOT); }

"\(" { return (LP); }

"\)" { return (RP); }

"\[" { return (LB); }

"\]" { return (RB); }

"\{" { return (LR); }

"\}" { return (RR); }

"\:" { return (COLON); }

"\." { return (PERIOD); }

"\," { return (COMMA); }

"\!" { return (EXCL); }

"\\*" { return (STAR); }

"\/" { return (SLASH); }

"\%" { return (PERCENT); }

"\&" { return (AMP); }

"\;" { return (SEMICOLON); }

"\+" { return (PLUS); }

"\-" { return (MINUS); }

"\=" { return (ASSIGN); }

"\~" { return (NOT); }

"\^" { return (XOR); }

"\|" { return (BAR); }

"\?" { return (QUESTION); }

"const" { return (CONST\_SYM); }

{digit}+ { yylval = atoi(yytext); return (INTEGER\_CONSTANT); }

{digit}+\.{digit}+ { yylval = (long)makeString(yytext); return (FLOAT\_CONSTANT); }

{letter}({letter}|{digit})\* { return (checkIdentifier(yytext)); }

\"([^"\n]|\\["\n])\*\" { yylval = (long)makeString(yytext); return (STRING\_LITERAL); }

\'([^'\n]|\'\')\' { yylval = \*(yytext + 1); return (CHARACTER\_CONSTANT); }

\/\\*([^\*]|\\*+[^\*/])\*\\*\/ { }

"//"[^\n]\* { }

%%

char \*makeString(char \*s)

{

char \*tmp;

tmp = malloc(strlen(s) + 1);

strcpy(tmp, s);

return (tmp);

}

int checkIdentifier(char \*s)

{

char \*table[] = {"int", "float", "char", "void"};

for (int i = 0; i < 4; i++)

{

if(strcmp(s, table[i]) == 0)

{

yylval = makeString(s);

return (TYPE\_IDENTIFIER);

}

}

yylval = \*s;

return (IDENTIFIER);

}

* Yacc

%{

#define YYSTYPE\_IS\_DECLARED 1

typedef long YYSTYPE;

#include "type.h"

#include "func.h"

#include <stdio.h>

#include <stdlib.h>

int yyerror(char\*);

int yylex();

%}

%token IDENTIFIER TYPE\_IDENTIFIER AUTO\_SYM BREAK\_SYM CASE\_SYM CONTINUE\_SYM DEFAULT\_SYM DO\_SYM ELSE\_SYM ENUM\_SYM FOR\_SYM IF\_SYM RETURN\_SYM SIZEOF\_SYM STATIC\_SYM STRUCT\_SYM SWITCH\_SYM TYPEDEF\_SYM UNION\_SYM WHILE\_SYM GOTO\_SYM

PLUSPLUS MINUSMINUS ARROW LSS GTR LEQ GEQ EQL NEQ AMPAMP BARBAR LSH RSH DOTDOTDOT LP RP LB RB LR RR COLON PERIOD COMMA EXCL STAR SLASH PERCENT AMP SEMICOLON PLUS MINUS ASSIGN NOT XOR BAR QUESTION INTEGER\_CONSTANT FLOAT\_CONSTANT STRING\_LITERAL CHARACTER\_CONSTANT CONST\_SYM

%start program

%%

program : translation\_unit { root=makeNode(N\_PROGRAM, NIL, $1, NIL); checkForwardReference(); }

;

translation\_unit : external\_declaration { $$=$1; }

| translation\_unit external\_declaration { $$=linkDeclaratorList($1, $2); }

;

external\_declaration : function\_definition { $$=$1; }

| declaration { $$=$1; }

;

function\_definition : declaration\_specifiers declarator { $$=setFunctionDeclaratorSpecifier($2, $1); } compound\_statement { $$=setFunctionDeclaratorBody($3, $4); current\_id=$2; }

| declarator { $$=setFunctionDeclaratorSpecifier($1, makeSpecifier(int\_type, 0)); } compound\_statement { $$=setFunctionDeclaratorBody($2, $3); current\_id=$1; }

;

declaration : declaration\_specifiers init\_declarator\_list SEMICOLON { $$=setDeclaratorListSpecifier($2, $1); }

;

declaration\_specifiers : type\_specifier { $$=makeSpecifier($1, 0); }

| storage\_class\_specifier { $$=makeSpecifier(0, $1); }

| type\_specifier declaration\_specifiers { $$=updateSpecifier($2, $1, 0); }

| storage\_class\_specifier declaration\_specifiers {$$=updateSpecifier($2, 0, $1); }

;

storage\_class\_specifier : AUTO\_SYM { $$=S\_AUTO; } | STATIC\_SYM { $$=S\_STATIC; } | TYPEDEF\_SYM {$$=S\_TYPEDEF; }

;

init\_declarator\_list : init\_declarator { $$=$1; }

| init\_declarator\_list COMMA init\_declarator { $$=linkDeclaratorList($1, $3); }

;

init\_declarator : declarator {$$=$1;}

| declarator ASSIGN initializer {$$=setDeclaratorInit((A\_ID\*)$1, (A\_NODE\*)$3); }

;

type\_specifier : struct\_specifier {$$=$1;}

| enum\_specifier {$$=$1;}

| TYPE\_IDENTIFIER {$$=$1;}

;

struct\_specifier : struct\_or\_union IDENTIFIER {$$=setTypeStructOrEnumIdentifier($1, $2, ID\_STRUCT); } LR { $$=current\_id; current\_level++; } struct\_declaration\_list RR {checkForwardReference(); $$=setTypeField($3, $6); current\_level--; current\_id=$5; }

| struct\_or\_union {$$=makeType($1); } LR {$$=current\_id; current\_level++; } struct\_declaration\_list RR {checkForwardReference(); $$=setTypeField($2, $5); current\_level--; current\_id=$4; }

| struct\_or\_union IDENTIFIER {$$=getTypeOfStructOrEnumRefIdentifier($1, $2, ID\_STRUCT); }

;

struct\_or\_union : STRUCT\_SYM {$$=T\_STRUCT; }

| UNION\_SYM {$$=T\_UNION; }

;

struct\_declaration\_list : struct\_declaration {$$=$1;}

| struct\_declaration\_list struct\_declaration {$$=linkDeclaratorList($1, $2); }

;

struct\_declaration : type\_specifier struct\_declarator\_list SEMICOLON {$$=setStructDeclaratorListSpecifier($2, $1); }

;

struct\_declarator\_list : struct\_declarator {$$=$1;}

| struct\_declarator\_list COMMA struct\_declarator {$$=linkDeclaratorList($1, $3);}

;

struct\_declarator : declarator {$$=$1;}

;

enum\_specifier : ENUM\_SYM IDENTIFIER {$$=setTypeStructOrEnumIdentifier(T\_ENUM, $2, ID\_ENUM); } LR enumerator\_list RR {$$=setTypeField($3, $5); }

| ENUM\_SYM {$$=makeType(T\_ENUM);} LR enumerator\_list RR {$$=setTypeField($2, $4);}

| ENUM\_SYM IDENTIFIER {$$=getTypeOfStructOrEnumRefIdentifier(T\_ENUM, $2, ID\_ENUM); }

;

enumerator\_list : enumerator {$$=$1;}

| enumerator\_list COMMA enumerator {$$=linkDeclaratorList($1, $3);}

;

enumerator : IDENTIFIER {$$=setDeclaratorKind(makeIdentifier($1), ID\_ENUM\_LITERAL);}

| IDENTIFIER {$$=setDeclaratorKind(makeIdentifier($1), ID\_ENUM\_LITERAL);} ASSIGN constant\_expression {$$=setDeclaratorInit($2, $4);}

;

declarator : pointer direct\_declarator {$$=setDeclaratorElementType($2, $1);}

| direct\_declarator {$$=$1;}

;

constant\_expression\_opt : /\* empty \*/ {$$=NIL;}

| constant\_expression {$$=$1;}

;

parameter\_type\_list\_opt : /\* empty \*/ {$$=NIL;}

| parameter\_type\_list {$$=$1;}

;

pointer : STAR {$$=makeType(T\_POINTER);}

| STAR pointer {$$=setTypeElementType($2, makeType(T\_POINTER));}

;

direct\_declarator : IDENTIFIER {$$=makeIdentifier($1);}

| LP declarator RP {$$=$2;}

| direct\_declarator LB constant\_expression\_opt RB {$$=setDeclaratorElementType($1, setTypeExpr(makeType(T\_ARRAY), $3));}

| direct\_declarator LP {$$=current\_id; current\_level++;} parameter\_type\_list\_opt RP {checkForwardReference(); current\_id=$3; current\_level--; $$=setDeclaratorElementType($1, setTypeField(makeType(T\_FUNC), $4));}

;

parameter\_type\_list : parameter\_list {$$=$1;}

| parameter\_list COMMA DOTDOTDOT {$$=linkDeclaratorList($1, setDeclaratorKind(makeDummyIdentifier(), ID\_PARM));}

;

parameter\_list : parameter\_declaration {$$=$1;}

| parameter\_list COMMA parameter\_declaration {$$=linkDeclaratorList($1, $3);}

;

parameter\_declaration : declaration\_specifiers declarator {$$=setParameterDeclaratorSpecifier($2,$1);}

| declaration\_specifiers abstract\_declarator\_opt {$$=setParameterDeclaratorSpecifier(setDeclaratorType(makeDummyIdentifier(), $2), $1);}

;

abstract\_declarator\_opt : /\* empty \*/ {$$=NIL;}

| abstract\_declarator {$$=$1;}

;

abstract\_declarator : pointer {$$=makeType(T\_POINTER);}

| direct\_abstract\_declarator {$$=$1;}

| pointer direct\_abstract\_declarator {$$=setTypeElementType($2, makeType(T\_POINTER));}

;

direct\_abstract\_declarator : LP abstract\_declarator RP {$$=$2;}

| LB constant\_expression\_opt RB {$$=setTypeExpr(makeType(T\_ARRAY), $2);}

| LP parameter\_type\_list\_opt RP {$$=setTypeExpr(makeType(T\_FUNC), $2);}

| direct\_abstract\_declarator LB constant\_expression\_opt RB {$$=setTypeElementType($1, setTypeExpr(makeType(T\_ARRAY), $3));}

| direct\_abstract\_declarator LP parameter\_type\_list\_opt RP {$$=setTypeElementType($1, setTypeExpr(makeType(T\_FUNC),$3));}

initializer : constant\_expression {$$=(A\_NODE\*)makeNode(N\_INIT\_LIST\_ONE, NIL, $1, NIL);}

| LR initializer\_list RR {$$=$2;}

| LR initializer\_list COMMA RR {$$=$2;}

;

initializer\_list : initializer {$$=makeNode(N\_INIT\_LIST, $1, NIL, makeNode(N\_INIT\_LIST\_NIL, NIL, NIL, NIL));}

| initializer\_list COMMA initializer {$$=makeNodeList(N\_INIT\_LIST,$1,$3);}

;

statement : labeled\_statement {$$=$1;}

| compound\_statement {$$=$1;}

| expression\_statement {$$=$1;}

| selection\_statement {$$=$1;}

| iteration\_statement {$$=$1;}

| jump\_statement {$$=$1;}

;

labeled\_statement : CASE\_SYM constant\_expression COLON statement {$$=makeNode(N\_STMT\_LABEL\_CASE, $2, NIL, $4);}

| DEFAULT\_SYM COLON statement {$$=makeNode(N\_STMT\_LABEL\_DEFAULT, NIL, $3, NIL);}

;

compound\_statement : LR {$$=current\_id; current\_level++; } declaration\_list statement\_list RR {checkForwardReference(); $$=makeNode(N\_STMT\_COMPOUND, $3, NIL, $4); current\_id=$2; current\_level--;}

;

declaration\_list : /\* empty \*/ {$$=NIL;}

| declaration\_list declaration {$$=linkDeclaratorList($1, $2);}

;

statement\_list : /\* empty \*/ {$$=NIL;}

| statement\_list statement {$$=makeNodeList(N\_STMT\_LIST, $1, $2);}

;

expression\_statement : SEMICOLON {$$=makeNode(N\_STMT\_EMPTY, NIL, NIL, NIL);}

| expression SEMICOLON {$$=makeNode(N\_STMT\_EXPRESSION, NIL, $1, NIL);}

selection\_statement : IF\_SYM LP expression RP statement {$$=makeNode(N\_STMT\_IF, $3, NIL, $5);}

| IF\_SYM LP expression RP statement ELSE\_SYM statement {$$=makeNode(N\_STMT\_IF\_ELSE, $3, $5, $7);}

| SWITCH\_SYM LP expression RP statement {$$=makeNode(N\_STMT\_SWITCH, $3, NIL, $5);}

;

iteration\_statement : WHILE\_SYM LP expression RP statement {$$=makeNode(N\_STMT\_WHILE, $3, NIL, $5);}

| DO\_SYM statement WHILE\_SYM LP expression RP SEMICOLON {$$=makeNode(N\_STMT\_DO, $2, NIL, $5);}

| FOR\_SYM LP expression\_opt SEMICOLON expression\_opt SEMICOLON expression\_opt RP statement {$$=makeNode(N\_STMT\_FOR, $3, NIL, $5);}

;

expression\_opt : /\* empty \*/ {$$=NIL;}

| expression {$$=$1;}

;

jump\_statement : RETURN\_SYM expression\_opt SEMICOLON {$$=makeNode(N\_STMT\_RETURN, NIL, $2, NIL);}

| CONTINUE\_SYM SEMICOLON {$$=makeNode(N\_STMT\_CONTINUE, NIL, NIL, NIL);}

| BREAK\_SYM SEMICOLON {$$=makeNode(N\_STMT\_BREAK, NIL, NIL, NIL);}

;

primary\_expression : IDENTIFIER {$$=makeNode(N\_EXP\_IDENT, NIL, getIdentifierDeclared($1), NIL);}

| INTEGER\_CONSTANT {$$=makeNode(N\_EXP\_INT\_CONST, NIL, $1, NIL);}

| FLOAT\_CONSTANT {$$=makeNode(N\_EXP\_FLOAT\_CONST, NIL, $1, NIL);}

| CHARACTER\_CONSTANT {$$=makeNode(N\_EXP\_CHAR\_CONST, NIL, $1, NIL);}

| STRING\_LITERAL {$$=makeNode(N\_EXP\_STRING\_LITERAL, NIL, $1, NIL);}

| LP expression RP {$$=$2;}

;

postfix\_expression : primary\_expression {$$=$1;}

| postfix\_expression LB expression RB {$$=makeNode(N\_EXP\_ARRAY, $1, NIL, $3);}

| postfix\_expression LP arg\_expression\_list\_opt RP {$$=makeNode(N\_EXP\_FUNCTION\_CALL, $1, NIL, $3);}

| postfix\_expression PERIOD IDENTIFIER {$$=makeNode(N\_EXP\_STRUCT, $1, NIL, $3);}

| postfix\_expression ARROW IDENTIFIER {$$=makeNode(N\_EXP\_ARROW, $1, NIL, $3);}

| postfix\_expression PLUSPLUS {$$=makeNode(N\_EXP\_POST\_INC, NIL, $1, NIL);}

| postfix\_expression MINUSMINUS {$$=makeNode(N\_EXP\_POST\_DEC, NIL, $1, NIL);}

;

arg\_expression\_list\_opt : /\* empty \*/ {$$=makeNode(N\_ARG\_LIST\_NIL, NIL, NIL, NIL);}

| arg\_expression\_list {$$=$1;}

;

arg\_expression\_list : assignment\_expression {$$=makeNode(N\_ARG\_LIST, $1, NIL, makeNode(N\_ARG\_LIST\_NIL, NIL, NIL, NIL));}

| arg\_expression\_list COMMA assignment\_expression {$$=makeNodeList(N\_ARG\_LIST, $1, $3);}

;

unary\_expression : postfix\_expression {$$=$1;}

| PLUSPLUS unary\_expression {$$=makeNode(N\_EXP\_PRE\_INC,NIL,$2,NIL);}

| MINUSMINUS unary\_expression {$$=makeNode(N\_EXP\_PRE\_DEC,NIL,$2,NIL);}

| AMP cast\_expression {$$=makeNode(N\_EXP\_AMP,NIL,$2,NIL);}

| STAR cast\_expression {$$=makeNode(N\_EXP\_STAR,NIL,$2,NIL);}

| EXCL cast\_expression {$$=makeNode(N\_EXP\_NOT,NIL,$2,NIL);}

| MINUS cast\_expression {$$=makeNode(N\_EXP\_MINUS,NIL,$2,NIL);}

| PLUS cast\_expression {$$=makeNode(N\_EXP\_PLUS,NIL,$2,NIL);}

| SIZEOF\_SYM unary\_expression {$$=makeNode(N\_EXP\_SIZE\_EXP,NIL,$2,NIL);}

| SIZEOF\_SYM LP type\_name RP {$$=makeNode(N\_EXP\_SIZE\_TYPE,NIL,$3,NIL);}

;

cast\_expression : unary\_expression {$$=$1;}

| LP type\_name RP cast\_expression {$$=makeNode(N\_EXP\_CAST,$2,NIL, $4);}

;

type\_name : declaration\_specifiers abstract\_declarator {$$=setTypeNameSpecifier($2,$1);}

;

multiplicative\_expression : cast\_expression {$$=$1;}

| multiplicative\_expression STAR cast\_expression {$$=makeNode(N\_EXP\_MUL, $1, NIL, $3);}

| multiplicative\_expression SLASH cast\_expression {$$=makeNode(N\_EXP\_DIV, $1, NIL, $3);}

| multiplicative\_expression PERCENT cast\_expression {$$=makeNode(N\_EXP\_MOD, $1, NIL, $3);}

;

additive\_expression : multiplicative\_expression {$$=$1;}

| additive\_expression PLUS multiplicative\_expression {$$=makeNode(N\_EXP\_ADD,$1,NIL,$3);}

| additive\_expression MINUS multiplicative\_expression {$$=makeNode(N\_EXP\_SUB,$1,NIL,$3);}

;

shift\_expression : additive\_expression {$$=$1;}

;

relational\_expression : shift\_expression {$$=$1;}

| relational\_expression LSS shift\_expression {$$=makeNode(N\_EXP\_LSS, $1, NIL, $3);}

| relational\_expression GTR shift\_expression {$$=makeNode(N\_EXP\_GTR, $1, NIL, $3);}

| relational\_expression LEQ shift\_expression {$$=makeNode(N\_EXP\_LEQ, $1, NIL, $3);}

| relational\_expression GEQ shift\_expression {$$=makeNode(N\_EXP\_GEQ, $1, NIL, $3);}

;

equality\_expression : relational\_expression {$$=$1;}

| equality\_expression EQL relational\_expression {$$=makeNode(N\_EXP\_EQL,$1,NIL,$3);}

| equality\_expression NEQ relational\_expression {$$=makeNode(N\_EXP\_NEQ,$1,NIL,$3);}

;

AND\_expression : equality\_expression {$$=$1;}

;

exclusive\_OR\_expression : AND\_expression {$$=$1;}

;

inclusive\_OR\_expression : exclusive\_OR\_expression {$$=$1;}

| inclusive\_OR\_expression BAR exclusive\_OR\_expression {$$=makeNode(N\_EXP\_OR, $1, NIL, $3);}

;

logical\_AND\_expression : inclusive\_OR\_expression {$$=$1;}

| logical\_AND\_expression AMPAMP inclusive\_OR\_expression {$$=makeNode(N\_EXP\_AND,$1,NIL, $3);}

;

logical\_OR\_expression : logical\_AND\_expression {$$=$1;}

| logical\_OR\_expression BARBAR logical\_AND\_expression {$$=makeNode(N\_EXP\_OR, $1, NIL, $3);}

;

conditional\_expression : logical\_OR\_expression {$$=$1;}

;

assignment\_expression : conditional\_expression {$$=$1;}

| unary\_expression ASSIGN assignment\_expression {$$=makeNode(N\_EXP\_ASSIGN, $1, NIL, $3);}

;

comma\_expression : assignment\_expression {$$=$1;}

;

expression : comma\_expression {$$=$1;}

;

constant\_expression : assignment\_expression {$$=$1;}

;

%%

extern int syntax\_err;

extern A\_NODE \*root;

void main() {

initialize();

yyparse();

if (syntax\_err) exit(1);

print\_ast(root);

printf("success!\n");

exit(0);

}

extern char \*yytext;

int yyerror(char \*s) { printf("%s near %s\n", s, yytext); exit(1); }

int yywrap() { return (1); }