A3 Yacc Specification: parse.y

```
%{
#include "type.h"
extern int line_no, syntax_err;
extern A_NODE *root;
extern A_ID *current_id;
extern int current_level;
extern A_TYPE *int_type;
%}
%start program
%token IDENTIFIER TYPE_IDENTIFIER FLOAT_CONSTANT INTEGER_CONSTANT
        CHARACTER_CONSTANT STRING_LITERAL PLUS MINUS PLUSPLUS
        MINUSMINUS BAR AMP BARBAR AMPAMP ARROW
        SEMICOLON LSS GTR LEQ GEQ EQL NEQ DOTDOTDOT
        LP RP LB RB LR RR PERIOD COMMA EXCL STAR SLASH PERCENT ASSIGN
        COLON AUTO_SYM STATIC_SYM TYPEDEF_SYM
        STRUCT_SYM ENUM_SYM SIZEOF_SYM UNION_SYM
        IF_SYM ELSE_SYM WHILE_SYM DO_SYM FOR_SYM CONTINUE_SYM
        BREAK_SYM RETURN_SYM SWITCH_SYM CASE_SYM DEFAULT_SYM
%%
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_list_opt
        :                               {$$=NIL;}
        | declaration_list              {$$=$1;}
        ;
declaration_list
        : declaration                   {$$=$1;}
        | declaration_list declaration  {$$=linkDeclaratorList($1,$2);}
        ;
declaration
        : declaration_specifiers init_declarator_list_opt 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_opt
        :                               {$$=makeDummyIdentifier();}
        | init_declarator_list          {$$=$1;}
        ;
init_declarator_list
```

```
        : init_declarator                                   {$$=$1;}
        | init_declarator_list COMMA init_declarator
                                                            {$$=linkDeclaratorList($1,$3);}
        ;
init_declarator
        : declarator                                        {$$=$1;}
        | declarator ASSIGN initializer                     {$$=setDeclaratorInit($1,$3);}
        ;
initializer
        : constant_expression       {$$=makeNode(N_INIT_LIST_ONE,NIL,$1,NIL);}
        | LR initializer_list RR    {$$=$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);}
        ;
type_specifier
        : struct_type_specifier                             {$$ = $1;}
        | enum_type_specifier                               {$$ = $1;}
        | TYPE_IDENTIFIER                                   {$$ = $1;}
        ;
struct_type_specifier
        : struct_or_union IDENTIFIER
          {$$=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_type_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 expression {$$=setDeclaratorInit($2,$4);}
        ;
declarator
        : pointer direct_declarator          {$$=setDeclaratorElementType($2,$1);}
        | direct_declarator                  {$$=$1;}
        ;
pointer
        : STAR              {$$=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_opt
        :                                    {$$=NIL;}
        | parameter_type_list                {$$=$1;}
        ;
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
        :                                       {$$=NIL;}
        | abstract_declarator                   {$$=$1;}
        ;
abstract_declarator
        : direct_abstract_declarator            {$$=$1;}
        | pointer                               {$$=makeType(T_POINTER);}
        | pointer direct_abstract_declarator
                        {$$=setTypeElementType($2,makeType(T_POINTER));}
        ;
direct_abstract_declarator
        : LP abstract_declarator RP             {$$=$2;}
        | LB constant_expression_opt RB
         {$$=setTypeExpr(makeType(T_ARRAY),$2);}
        | direct_abstract_declarator LB constant_expression_opt RB
         {$$=setTypeElementType($1,setTypeExpr(makeType(T_ARRAY),$3));}
        | LP parameter_type_list_opt RP
         {$$=setTypeExpr(makeType(T_FUNC),$2);}
        | direct_abstract_declarator LP parameter_type_list_opt RP
         {$$=setTypeElementType($1,setTypeExpr(makeType(T_FUNC),$3));}
        ;
statement_list_opt
        :                               {$$=makeNode(N_STMT_LIST_NIL,NIL,NIL,NIL);}
```

```
        | statement_list              {$$=$1;}
        ;
statement_list
        : statement {$$=makeNode(N_STMT_LIST,$1,NIL,
          makeNode(N_STMT_LIST_NIL,NIL,NIL,NIL));}
        | statement_list statement {$$=makeNodeList(N_STMT_LIST,$1,$2);}
        ;
statement
        : labeled_statement                 {$$=$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_opt
          statement_list_opt RR {checkForwardReference();
          $$=makeNode(N_STMT_COMPOUND,$3,NIL,$4); current_id=$2;
          current_level--;}
        ;
expression_statement
        : SEMICOLON                 {$$=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 for_expression RP statement
                {$$=makeNode(N_STMT_FOR,$3,NIL,$5);}
        ;
for_expression
        : expression_opt SEMICOLON expression_opt SEMICOLON expression_opt
                {$$=makeNode(N_FOR_EXP,$1,$3,$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
                {$$=makeNode(N_STMT_BREAK,NIL,NIL,NIL);}
        ;
arg_expression_list_opt
        :                       {$$=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);}
        ;
constant_expression_opt
        :                                   {$$=NIL;}
        | constant_expression               {$$=$1;}
        ;
constant_expression
        : expression                        {$$=$1;}
        ;
expression
        : comma_expression                  {$$=$1;}
        ;
comma_expression
        : assignment_expression             {$$=$1;}
        ;
assignment_expression
        : conditional_expression            {$$=$1;}
        | unary_expression ASSIGN assignment_expression
                                            {$$=makeNode(N_EXP_ASSIGN,$1,NIL,$3);}
        ;
conditional_expression
        : logical_or_expression             {$$=$1;}
        ;
logical_or_expression
        : logical_and_expression            {$$=$1;}
        | logical_or_expression BARBAR logical_and_expression
                                            {$$=makeNode(N_EXP_OR,$1,NIL,$3);}
        ;
logical_and_expression
        : bitwise_or_expression             {$$=$1;}
        | logical_and_expression AMPAMP bitwise_or_expression
```

```
                                        {$$=makeNode(N_EXP_AND,$1,NIL,$3);}
        ;
bitwise_or_expression
        : bitwise_xor_expression        {$$=$1;}
        ;
bitwise_xor_expression
        : bitwise_and_expression        {$$=$1;}
        ;
bitwise_and_expression
        : equality_expression           {$$=$1;}
        ;
equality_expression
        : relational_expression         {$$=$1;}
        | equality_expression EQL relational_expression
                                        {$$=makeNode(N_EXP_EQL,$1,NIL,$3);}
        | equality_expression NEQ relational_expression
                                        {$$=makeNode(N_EXP_NEQ,$1,NIL,$3);}
        ;
relational_expression
        : shift_expression              {$$=$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);}
        ;
shift_expression
        : additive_expression           {$$=$1;}
        ;
additive_expression
        : multiplicative_expression     {$$=$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);}
        ;
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);}
        ;
cast_expression
        : unary_expression              {$$=$1;}
        | LP type_name RP cast_expression
                                        {$$=makeNode(N_EXP_CAST,$2,NIL,$4);}
        ;
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);}
```

```
        ;
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
                                    {$$=makeNode(N_EXP_POST_DEC,NIL,$1,NIL);}
        ;
primary_expression
        : IDENTIFIER
          {$$=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        {$$=$2;}
        ;
type_name
        : declaration_specifiers  abstract_declarator_opt
                                    {$$=setTypeNameSpecifier($2,$1);}
        ;
%%
extern  char  *yytext;
yyerror(char  *s)
{
        syntax_err++;
```

```
        printf("line %d: %s near %s ₩n", line_no, s,yytext);
}
```