带数组与字符串类型的程序语言的词法分析与语法分析实验报告

齐思远，宋士祥，苏唐俊

December 2023 - January 2024

1 要求 1：数组相关的变量声明与表达式

1.1 需要考虑的程序语句

首先，我们需要明确需要解析什么样的程序语句，分一维和多维数组分开叙述：

1.1.1 一维数组

具体来说，我们需要考虑如下三个方面：

(1) **数组的声明:** 参考 C 语言，我们可以以变量名后面的中括号来识别数组。根据和老师的讨论可知，只需要考虑指定常数长度的数组。故需要解析的指令包括：

```
var a[3];
```

(2) **数组的相关表达式:** 数组取索引后，就是一个值，应该支持这些值参与表达式计算。比如，需要解析的指令包括如下内容等等：

```
a[2] = a[0] + a[3];
```

(3) **指针形式的数组:**C 语言中，数组和指针常常联系在一起。创建一个指针，可以让指针指向一段连续的内存，从而实现了数组的功能。所以我们添加了以指针类型声明数组的功能，以及用解引用的方式参与表达式计算，下列指令展示了一些需要解析的内容：

```
var *a;
*a = *(a + 4) + *(a + 8);
```

1.1.2 多维数组

与一维数组相似，我们只需要把维度从一维拓展到多维，此外还要支持多级指针。如下是一些例子：

```
var a[1][2];
var a[3][2][2];
a[2][3] = 1;
a[0][1] = a[0][2] * a[3][0];
var ***abc;
*a = *(a + 4) * *******(a + 8);
```

1.2 相关指令的词法、语法分析实现

1.2.1 lang.l 文件的改动

由于并未涉及新的标识符，故本文件无需改动。

1.2.2 lang.h 文件的改动

关于表达式类型，在 ExprType 中，我们需要新添一个类型 T_ARRAY，用以表示数组类型；关于指令类型，在枚举类型 CmdType 中，我们需要新添 T_ARR_DECL 以及 T_PTR_DECL，分别用以数组和指针类型的初始化；相应的，在 expr 和 cmd 结构体中，要为新添加的类型提供相应的字段。

为了支持上述内容，还要设计数组和指针的构造函数，这既包括表达式的构造，也包括声明指令的解析。下列代码展示了上述所说的新添内容：

```
enum ExprType {
    //...之前的内容
    // 数组类型
    T_ARRAY,
    // 多维数组类型类型
    T_MULTI_ARRAY
};

enum CmdType {
  //...之前的内容
  // 数组声明指令
  T_ARR_DECL,
  // 指针声明指令
  T_PTR_DECL,
};

struct expr {
  enum ExprType t;
  union {
    //...之前的内容
    // 数组类型字段
    struct { char * array; struct expr * index; } ARRAY;
    // 多维数组字段
    struct {
        struct expr *array;
        struct expr *index;
    } multi_array;
  } d;
};

struct cmd {
  enum CmdType t;
  union {
    //...之前的内容
    // 用于声明数组
    struct { char * name; struct expr * size; struct init_list * init_expr;} ARR_DECL;
    // 用于声明指针类型
    struct { char * name; int ptr_level; struct expr * size;} PTR_DECL;
  } d;
};

// 用于构造数组
struct cmd * TArrDecl(char * name, struct expr * size, struct init_list * init_expr);

// 用于构造指针
struct cmd * TPtrDecl(char * name, int ptr_level, struct expr * size);
```

```
// 用于构造数组和多维数组
struct expr * TArray(char * array, struct expr * index);
struct expr *TMultiArray(struct expr *array, struct expr *index);
```

1.2.3 lang.c 文件的改动

在该文件中，要给出 `lang.h` 中声明的函数的具体实现，以实现其具体功能。由于函数的功能已经在 `lang.h` 中说明，本部分内容请见附录。

1.2.4 lang.y 文件的改动

在该文件中，需要在开头的 `union` 中添加一个字段 `sizes`，用于表示多维数组的索引。此外，要添加一些非终结符，如指针层级、多维数组索引，并完成相关产生式的解析。具体实现如下：

```
%union {
// ...之前的内容
// 多维数组索引字段
struct expr_list *sizes;
}

// 指针级数
%type <n> NT_PTR_LEVEL

// 多维数组索引
%type <sizes> NT_MULTI_ARR_SIZES

// 用于解析指针层级
NT_PTR_LEVEL:
  // 最初是1
  TM_MUL
  {
    $$ = 1;
  }
  // 每多一级，递增1
  | TM_MUL NT_PTR_LEVEL
  {
    $$ = $2 + 1;
  }
;

// 用于解析多维数组索引部分
NT_MULTI_ARR_SIZES:
// 明确说明大小情形
TM_LEFT_BRACKET NT_EXPR TM_RIGHT_BRACKET
  {
    $$ = create_expr_list($2);
  }
// 新维度，说明了大小
| NT_MULTI_ARR_SIZES TM_LEFT_BRACKET NT_EXPR TM_RIGHT_BRACKET
  {
    $$ = append_to_expr_list($1, $3);
  }
;
```

```
40
// 变量的声明，包含了单语句多变量，也包含数组，指针等等
42 NT_VAR_DECL:
// 单个变量声明，无初值
44   TM_IDENT
   {
46     $$ = (struct var_decl){ .name = $1, .type = VAR_SIMPLE, .sizes = NULL, .ptr_level = 0 };
   }
48 // 数组(一维或多维)声明，无初值
 | TM_IDENT NT_MULTI_ARR_SIZES
50   {
     $$ = (struct var_decl){ .name = $1, .type = VAR_ARRAY, .sizes = $2, .ptr_level = 0 };
52   }
// 指针类型，一级或多级指针
54 | NT_PTR_LEVEL TM_IDENT
   {
56     $$ = (struct var_decl){ .name = $2, .type = VAR_POINTER, .sizes = NULL, .ptr_level = $1 };
   }
58 ;

60 NT_EXPR_2:
// ...之前的内容
62 // 多维数组情形
 | NT_EXPR_2 TM_LEFT_BRACKET NT_EXPR TM_RIGHT_BRACKET
64   {
     $$ = (TMultiArray($1, $3)); // 多维数组访问
66   }
 ;
```

其中变量、数组、指针的声明涉及到了单语句多变量声明，详细请见要求 3 部分的说明。此外，多维数组的所以部分也涉及到了表达式列表相关内容，会在要求 2 中说明。

2 要求 2：变量和数组声明的同时初始化

2.1 需要考虑的程序语句

2.1.1 变量声明的同时初始化

变量初始化时的值比较多样，可以是整数、其他变量、数组取索引等等，实现时只需将等号右侧设为表达式类型即可。如下是一些需要考虑的例子：

```
1 var a = 3;
  var b = a;
3 var c = 3 + 4;
  var d = b * c;
5 var e = a[2];
  var hello = world[2][3];
7 var f = *(a + 4);
```

2.1.2 数组及指针声明的同时初始化

参照 C 语言，数组的声明初始化主要分为用大括号形式的数组声明，以及用关键字 `malloc` 动态分配内存。如下给出了我们设计时考虑的例子：

```
var a[3] = {2, 3, 4};
var mul[2][2] = {{1,2}, {3,4}};
var *a = malloc(n)
```

2.2 相关指令的词法、语法分析实现

2.2.1 lang.l 文件的改动

在该要求中，仍无需改动

2.2.2 lang.h 文件的改动

为了支持对数组的初始化,需要定义具体数组的结构体,为了便于编程,我们分别设计了 `expr_list` 和 `init_list`; 在表达式结果体中，添加初始化列表字段 `init_list`。

实现的最终版本中，带初始化和不带初始化的声明复用了同一函数（初始化值是否为 `NULL`），故相关的构造函数就是要求 1 中所呈现的。此外，还需要定义一些函数来创建数组以及向数组中添加元素。如下是具体代码实现:

```
// 表达式列表类型
struct expr_list {
    // 当前表达式的指针
    struct expr *expr;
    // 表达式列表的下一个结点指针
    struct expr_list *next;
};

// 初始化列表类型
struct init_list {
    // 指向表达式链表的指针
    struct expr_list *exprs;
};

struct expr {
  enum ExprType t;
  union {
    // ...其他内容
    // 初始化列表字段
    struct init_list *init_list;
  } d;
};

// 用于初始化表达式列表和声明初始化列表，以及向表达式中添加新的表达式
struct expr_list *create_expr_list(struct expr *expr);
struct init_list *create_init_list(struct expr_list *expr_list);
struct expr_list *append_to_expr_list(struct expr_list *list, struct expr *expr);
```

2.2.3 lang.c 文件的改动

对 `lang.h` 中新定义的函数的具体实现请见附录。

2.2.4 lang.y 文件的改动

在开头的 `union` 中，需要添加初始化列表、表达式列表字段，并添加这二者的非终结符，以及它们的展开式解析，并将其用于带初始化声明的表达式中。具体改动如下:

```
%union {
// ...其他内容
// 初始化列表字段
struct init_list *init_list;
// 表达式列表字段
struct expr_list *e_list;
}

// 初始化列表
%type <init_list> NT_INIT_LIST
// 表达式列表
%type <e_list> NT_EXPR_LIST

// 调用create_init_list函数，解析初始化列表
NT_INIT_LIST:
    TM_LEFT_BRACE NT_EXPR_LIST TM_RIGHT_BRACE
    {
        $$ = create_init_list($2);
    }
;

// 解析表达式列表
NT_EXPR_LIST:
// 空列表，返回NULL
    {
        $$ = NULL;
    }
// 单个表达式情形，调用create_expr_list创建列表
| NT_EXPR
    {
        $$ = create_expr_list($1);
    }
// 多个表达式情形，调用append_to_expr_list向其中添加新表达式
| NT_EXPR_LIST TM_COMMA NT_EXPR
    {
        $$ = append_to_expr_list($1, $3);
    }
// 本身就是初始化列表，直接调用create_expr_list函数即可
| NT_INIT_LIST
    {
        $$ = create_expr_list(new_expr(T_INIT_LIST, $1));
    }
// 表达式列表后跟随初始化列表，向其中添加即可
| NT_EXPR_LIST TM_COMMA NT_INIT_LIST
    {
        $$ = append_to_expr_list($1, new_expr(T_INIT_LIST, $3));
    }
;

// 变量的声明，包含了单语句多变量，也包含数组，指针等等
NT_VAR_DECL:
// ...其他内容
// 单个变量声明及初始化
```

```
| TM_IDENT TM_ASGNOP NT_EXPR
  {
    $$ = (struct var_decl){ .name = $1, .type = VAR_SIMPLE, .init_expr = $3 };
  }
// 数组(一维或多维)声明, 以及初始化
| TM_IDENT NT_MULTI_ARR_SIZES TM_ASGNOP NT_INIT_LIST
  {
    $$ = (struct var_decl){ .name = $1, .type = VAR_ARRAY, .sizes = $2, .ptr_level = 0, .
        init_expr_list = $4 };
  }
;

// 解析指令类型
NT_CMD:
// ...其他内容
// 解析动态数组初始化, malloc
| TM_VAR NT_PTR_LEVEL TM_IDENT TM_ASGNOP TM_MALLOC TM_LEFT_PAREN NT_EXPR TM_RIGHT_PAREN
  {
    // $2 是指针级别
    $$ = TPtrDecl($3, $2, $7);
  }
```

3 要求 3：字符串常量和单语句多变量声明

3.1 需要考虑的程序语句

3.1.1 字符声明及初始化、字符串声明及字符相关表达式

为了实现字符，我们引入了新的关键字 `char`。考虑到只需实现字符串常量，采用长度为固定常数的数组形式来声明。此外也应该支持字符相关的表达式。如下列例子：

```
char a;
char a = 's';
char a[3];
a[1] = 's';
```

3.1.2 字符串声明初始化（双引号包围字符串及字符数组）

参照 C 语言，字符串的初始化主要包括两种方式，分别是利用双引号包围的字符串和字符数组，参照如下例子：

```
char a[3] = {'a', 'b', 'c'};
char a[5] = "hello";
char a[12] = "world";
```

3.1.3 单语句多变量声明

除了单语句声明多个普通变量，C 语音中还支持同时支持相应类型的数组、指针类型。故我们考虑了如下样例：

```
var a, b, c;
var a, b[2], *c;
var *a, **b, ***c;
var a = 1, b[3] = {2,3}, c[2][3] = {{1,2,3}, {4,5,6}}
```

3.2 相关指令的词法、语法分析实现

3.2.1 lang.l 文件的改动

在该要求中，需要解析 `char` 关键字；此外，字符数组中，需要其中都是单引号内一个单字符，故这种情况也需要解析；以及双引号包围的字符串也需要考虑。故新增内容如下：

```
"char" {
    return TM_CHAR;
}

'[_A-Za-z0-9]' {
    yylval.i = new_str(yytext, yyleng);
    return TM_SINGLE_CHAR;
}

\"[_A-Za-z][_A-Za-z0-9]*\"␣␣␣{
␣␣␣␣yylval.i␣=␣new_str(yytext,␣yyleng);
␣␣␣␣return␣TM_STRING;
}
```

3.2.2 lang.h 文件的改动

表达式类型中，需要新增字符常量和字符串常量；在指令类型中，需要新增字符变量的声明，两种不同初始化方式的字符串声明以及单语句多变量声明；为了支持单语句多变量，设计了辅助用的数据结构 `var_decl` 和 `var_decl_list`，以及相关的函数；以及与字符串相关的构造函数等等。具体设计如下：

```
enum ExprType {
  // ...其他内容
  // 字符常量类型
  T_CHAR,
  // 字符串常量
  T_STRING,
};

enum CmdType {
  // 字符变量声明
  T_CHAR_DECL,
  // 用双引号包围的字符串初始化字符串指令
  T_STRING_DECL_STRING,
  // 用字符数组初始化字符串指令
  T_STRING_DECL_ARRAY,
  // 单语句多变量声明指令
  T_MULTI_VAR_DECL
};

// 为了实现单语句多变量声明，辅助用的数据结构
struct var_decl {
    char *name;
    enum { VAR_SIMPLE, VAR_ARRAY, VAR_POINTER } type;
    // 用于数组的大小
    struct expr_list *sizes;
    //用于指针的层数
    int ptr_level;
    // 用于存储多维数组的维度数量
```

```
    int dimensions;
    // 初始化表达式
    struct expr *init_expr;
    // 初始化表达式列表
    struct init_list *init_expr_list;
};

// 将多个var_decl类型作为结点连接为链表，形成单语句多变量声明列表
struct var_decl_list {
    struct var_decl var;
    struct var_decl_list *next;
};

// 用于单语句多变量声明的解析
struct cmd * TMultiVarDecl(struct var_decl_list *vars);
// 用于初始化一个多变量列表
struct var_decl_list *create_var_decl_list(struct var_decl var);
// 用于向多变量列表中添加新的变量
struct var_decl_list *append_to_var_decl_list(struct var_decl_list *list, struct var_decl var);

struct expr {
  enum ExprType t;
  union {
    // ...其他内容
    // 字符常量字段
    struct { char *c; } CHAR;
    // 字符串常量字段
    struct { char *str; } STRING;
  } d;
};

struct cmd {
  enum CmdType t;
  union {
    // ...其他内容
    // 字符变量声明
    struct { char * name; struct expr * init_expr; } CHAR_DECL;
    // 用于以双引号包围的字符串初始化字符串指令
    struct { char * name; struct expr * size; struct expr * init_expr;} STRING_DECL_STRING;
    // 用于以字符数组初始化字符串指令
    struct { char * name; struct expr * size; struct init_list * init_expr;} STRING_DECL_ARRAY;
    // 用于多变量声明指令
    struct { struct var_decl_list * vars;} MULTI_VAR_DECL;
  } d;
};

// 用于字符声明
struct cmd * TCharDecl(char * name, struct expr * init_expr);
// 用于构造字符串，分别以引号包围的字符串和字符列表初始化
struct cmd * TStringDecl_String(char * name, struct expr * size, struct expr * init_expr);
struct cmd * TStringDecl_Array(char * name, struct expr * size, struct init_list * init_expr);

// 用于构造单个字符
// char *是因为输入为"'a'"
```

```
struct expr * TChar(char *c);
// 用于构造字符串
struct expr * TString(char *str);
```

3.2.3 lang.c 文件的改动

在 lang.h 中定义的相关函数的具体实现详情见附录

3.2.4 lang.y 文件的改动

在 union 中，需要新增字符相关以及支持单语句多变量声明的字段；关于终结符，需要添加字符和字符串常量；关于非终结符，需要添加字符列表字段以及多变量声明列表字段；以及相关产生式的解析。具体实现如下：

```
%union {
// 单个字符字段
char *single_char;
// 多变量声明列表字段
struct var_decl_list *var_list;
// 变量声明字段
struct var_decl var_decl;
}

// 字符常量
%token <single_char> TM_SINGLE_CHAR
// 字符串常量
%token <i> TM_STRING

// 初始化字符列表
%type <init_list> NT_INIT_CHAR_LIST
// 字符列表
%type <e_list> NT_CHAR_LIST

// 变量声明
%type <var_decl> NT_VAR_DECL
// 单语句多变量声明列表
%type <var_list> NT_VAR_LIST

// 解析初始字符列表，复用create_init_list函数
NT_INIT_CHAR_LIST:
    TM_LEFT_BRACE NT_CHAR_LIST TM_RIGHT_BRACE
    {
        $$ = create_init_list($2);
    }
;

// 解析字符列表
NT_CHAR_LIST:
// 单个字符情形
TM_SINGLE_CHAR
{
    $$ = create_expr_list(TChar($1));
}
// 字符串情形
| NT_CHAR_LIST TM_COMMA TM_SINGLE_CHAR
```

```
{
    $$ = append_to_expr_list($1, TChar($3));
}
;

// 解析多变量列表
NT_VAR_LIST:
// 单个变量情形
  NT_VAR_DECL
  {
    $$ = create_var_decl_list($1);
  }
// 多变量，递归解析
| NT_VAR_LIST TM_COMMA NT_VAR_DECL
  {
    $$ = append_to_var_decl_list($1, $3);
  }
;

// 解析指令类型
NT_CMD:
// ...其他内容
| TM_CHAR TM_IDENT
  {
    $$ = TCharDecl($2, NULL); // 创建一个不带初始化的字符变量声明
  }
  | TM_CHAR TM_IDENT TM_ASGNOP TM_SINGLE_CHAR
  {
    $$ = TCharDecl($2, TChar($4)); // 创建一个带有初始化的字符变量声明
  }
// 解析字符串类型，规定了大小
| TM_CHAR TM_IDENT TM_LEFT_BRACKET NT_EXPR TM_RIGHT_BRACKET
  {
      $$ = TStringDecl_String($2, $4, NULL);
  }
// 解析字符串类型，规定了大小，并以双引号包围字符串初始化
| TM_CHAR TM_IDENT TM_LEFT_BRACKET NT_EXPR TM_RIGHT_BRACKET TM_ASGNOP TM_STRING
  {
      $$ = TStringDecl_String($2, $4, TString($7));
  }
// 解析字符串类型，规定了大小，并以字符列表初始化
| TM_CHAR TM_IDENT TM_LEFT_BRACKET NT_EXPR TM_RIGHT_BRACKET TM_ASGNOP NT_INIT_CHAR_LIST
  {
      $$ = TStringDecl_Array($2, $4, $7);
  }
// 变量的声明，单/多变量，进一步在NT_VAR_LIST中解析
| TM_VAR NT_VAR_LIST
  {
    $$ = TMultiVarDecl($2);
  }
;

NT_EXPR_2:
// ...其他内容
```

```
// 单字符情形
| TM_SINGLE_CHAR
  {
      $$ = TChar($1);
  }
// 字符串情形
| TM_STRING
  {
      $$ = TString($1);
  }
```

4 可视化

对于最初版本的代码，只能打印出一行一行的解析结果。在此基础上，通过控制递归层数来打印空格，可以将结果美观化、可视化。主要函数如下：

```
// 打印缩进，使语法树美观
void print_indent(int depth) {
    for (int i = 0; i < depth; i++) {
        printf("␣␣");
    }
}
```

在打印相关信息时，控制好 `depth`，可打印出较为美观的结果。

5 合法性检查

对于字符串类型，如果初始化字符串的长度超过了声明的大小，在 C 语言里会报错。于是我们对这种情况进行了检测，代码实现如下：

```
// lang.c
// 字符串声明情形(以双引号包围字符串初始化)
    case T_STRING_DECL_STRING:
        printf("STRING_DECL(%s)\n", c->d.STRING_DECL_STRING.name);
        if (c->d.STRING_DECL_STRING.init_expr != NULL) {
            unsigned int malloc_size = c->d.STRING_DECL_STRING.size->d.CONST.value;
            unsigned int init_size = strlen(c->d.STRING_DECL_STRING.init_expr->d.STRING.str) - 2;
            print_indent(depth + 1);
            printf("Malloc␣Size:\n");
            print_indent(depth + 2);
            printf("%d\n", malloc_size);
            print_indent(depth + 1);
            printf("Init␣String:\n");
            print_expr(c->d.STRING_DECL_STRING.init_expr, depth + 2);
            if(malloc_size < init_size)
                printf("Error:␣Array␣initialization␣exceeds␣declared␣size.\n");
            else
                printf("Legal␣init␣size\n");
        }
        break;
    // 字符串声明情形(以字符数组初始化)
    case T_STRING_DECL_ARRAY:
        printf("STRING_DECL(%s)\n", c->d.STRING_DECL_ARRAY.name);
```

```
        if (c->d.STRING_DECL_ARRAY.init_expr != NULL) {
            int malloc_size = c->d.STRING_DECL_ARRAY.size->d.CONST.value;
            print_indent(depth + 1);
            printf("Malloc Size:\n");
            print_indent(depth + 2);
            printf("%d\n", malloc_size);

            print_indent(depth + 1);
            printf("Init String:\n");
            int list_size = print_char_list(c->d.STRING_DECL_ARRAY.init_expr, depth + 2);
            if(malloc_size < list_size)
                printf("Error: Array initialization exceeds declared size.\n");
            else
                printf("Legal init size\n");
        }
        break;
```

考虑如下的测试集：

```
char a[3] = "hel";
char a[3] = "hello";
char a[3] = {'1', '2', '3'};
char a[3] = {'h', 'e', 'l', 'l', 'o'}
```

解析结果会显示第一个和第三个是合法的，而第二个和第四个会报错，输出初始化长度超出了声明长度。

6 结果展示

对于三个要求中的测试集以及合法化检查的测试集，分别放在了 `problem1-4.jtl`，得出结果详见 `result.md`：

附录

三个要求中涉及的函数在 `lang.c` 文件中的具体实现如下所示：

A 要求 1

```
/**
 * 创建一个表示单个数组访问的表达式
 *
 * @param array 表示数组的变量名
 * @param index 表示访问数组元素的索引表达式
 * @return 返回一个新的表达式指针，表示数组访问
 */
struct expr * TArray(char * array, struct expr * index) {
  struct expr * res = new_expr_ptr();
  res->t = T_ARRAY;
  res->d.ARRAY.array = array;
  res->d.ARRAY.index = index;
  return res;
}

/**
```

```
 * 创建一个数组声明命令
 *
 * @param name 表示数组的名称
 * @param size 表示数组的大小
 * @param init_expr 表示数组的初始化表达式列表
 * @return 返回一个新的命令指针，表示数组声明
 */
struct cmd * TArrDecl(char * name, struct expr * size, struct init_list * init_expr) {
  struct cmd * res = new_cmd_ptr();
  res->t = T_ARR_DECL;
  res->d.ARR_DECL.name = name;
  res->d.ARR_DECL.size = size;
  res->d.ARR_DECL.init_expr = init_expr;
  return res;
}

/**
 * 创建一个指针声明命令。
 *
 * @param name 指针变量的名称。
 * @param ptr_level 指针的级别(一级或多级)
 * @param size 如果是动态分配的指针，则表示分配的大小；如果为静态或未分配，则为NULL。
 * @return 返回一个新的命令指针，表示指针的声明。
 */
struct cmd * TPtrDecl(char * name, int ptr_level, struct expr * size) {
  struct cmd * res = new_cmd_ptr();
  res->t = T_PTR_DECL;
  res->d.PTR_DECL.name = name;
  res->d.PTR_DECL.ptr_level = ptr_level;
  if(size != NULL)
    res->d.PTR_DECL.size = TMalloc(size);
  else
    res->d.PTR_DECL.size = NULL;
  return res;
}

/**
 * 创建一个表示多维数组访问的表达式
 *
 * @param array 表示已经部分访问的数组表达式
 * @param index 表示当前维度的索引表达式
 * @return 返回一个新的表达式指针，表示多维数组的进一步访问
 */
struct expr *TMultiArray(struct expr *array, struct expr *index) {
    struct expr *res = new_expr_ptr();
    res->t = T_MULTI_ARRAY;
    res->d.multi_array.array = array;
    res->d.multi_array.index = index;
    return res;
}
```

B 要求 2

```c
/**
 * 创建一个表达式列表节点
 *
 * @param expr 表示要添加到列表中的表达式
 * @return 返回一个新的表达式列表节点，其中包含提供的表达式
 */
struct expr_list *create_expr_list(struct expr *expr) {
    struct expr_list *new_list = malloc(sizeof(struct expr_list));
    if (!new_list) {
        return NULL;
    }
    new_list->expr = expr;
    new_list->next = NULL;
    return new_list;
}

/**
 * 创建一个初始化列表节点
 *
 * @param expr_list 表示要添加到初始化列表中的表达式列表
 * @return 返回一个新的初始化列表节点，其中包含提供的表达式列表
 */
struct init_list *create_init_list(struct expr_list *expr_list) {
    struct init_list *new_list = malloc(sizeof(struct init_list));
    if (!new_list) {
        // 如果内存分配失败，返回NULL
        return NULL;
    }
    new_list->exprs = expr_list;
    return new_list;
}

/**
 * 将一个表达式附加到表达式列表的末尾
 *
 * @param list 表示要附加表达式的列表
 * @param expr 表示要附加的表达式
 * @return 返回更新后的表达式列表
 */
struct expr_list *append_to_expr_list(struct expr_list *list, struct expr *expr) {
    struct expr_list *current = list;
    while (current->next != NULL) {
        current = current->next;
    }
    struct expr_list *new_node = create_expr_list(expr);
    if (!new_node) {
        // 如果创建新节点失败，保持列表不变
        return list;
    }
    current->next = new_node;
    return list;
}
```

C 要求 3

```
/**
 * 创建一个字符串声明命令（使用双引号包围的字符串初始化）
 *
 * @param name 表示字符串变量的名称
 * @param size 表示字符串的大小
 * @param init_expr 表示字符串初始化的表达式(双引号包围字符串)
 * @return 返回一个新的命令指针，表示字符串声明
 */
struct cmd * TStringDecl_String(char * name, struct expr * size, struct expr * init_expr) {
  struct cmd * res = new_cmd_ptr();
  res->t = T_STRING_DECL_STRING;
  res->d.STRING_DECL_STRING.name = name;
  res->d.STRING_DECL_STRING.size = size;
  res->d.STRING_DECL_STRING.init_expr = init_expr;
  return res;
}

/**
 * 创建一个字符数组声明命令（使用字符列表初始化）
 *
 * @param name 表示字符数组的名称
 * @param size 表示数组的大小
 * @param init_expr 表示字符数组初始化的字符列表
 * @return 返回一个新的命令指针，表示字符数组声明
 */
struct cmd * TStringDecl_Array(char * name, struct expr * size, struct init_list * init_expr) {
  struct cmd * res = new_cmd_ptr();
  res->t = T_STRING_DECL_ARRAY;
  res->d.STRING_DECL_ARRAY.name = name;
  res->d.STRING_DECL_ARRAY.size = size;
  res->d.STRING_DECL_ARRAY.init_expr = init_expr;
  return res;
}

/**
 * 创建一个单字符常量表达式。
 *
 * @param c 表示字符常量。
 * @return 返回一个新的表达式指针，表示字符常量。
 */
struct expr * TChar(char *c) {
    struct expr * res = new_expr_ptr();
    res->t = T_CHAR;
    res->d.CHAR.c = c;
    return res;
}

/**
 * 创建一个字符串常量表达式。
```

```
 *
 * @param str 表示字符串常量。
 * @return 返回一个新的表达式指针，表示字符串常量。
 */
struct expr * TString(char *str) {
    struct expr * res = new_expr_ptr();
    res->t = T_STRING;
    res->d.STRING.str = str;
    return res;
}

// 实现TCharDecl函数
struct cmd * TCharDecl(char *name, struct expr *init_expr) {
    struct cmd *res = new_cmd_ptr();
    res->t = T_CHAR_DECL;
    res->d.CHAR_DECL.name = name;
    res->d.CHAR_DECL.init_expr = init_expr;
    return res;
}



/**
 * 创建一个单语句多变量声明命令。
 *
 * @param vars 指向变量声明列表的指针，列表中的每个元素都是一个变量声明。
 * @return 返回一个新的命令指针，表示多变量的声明。
 */
struct cmd * TMultiVarDecl(struct var_decl_list *vars) {
    struct cmd *res = new_cmd_ptr();
    res->t = T_MULTI_VAR_DECL;
    res->d.MULTI_VAR_DECL.vars = vars;
    return res;
}


/**
 * 创建一个新的变量声明列表节点。
 *
 * @param var 变量声明的结构体。
 * @return 返回一个新的变量声明列表节点。
 */
struct var_decl_list *create_var_decl_list(struct var_decl var) {
    struct var_decl_list *new_node = malloc(sizeof(struct var_decl_list));
    if (new_node == NULL) {
        printf("Failure in malloc.\n");
        exit(0);
    }
    new_node->var = var;
    new_node->next = NULL;
    return new_node;
}

```

```
104 /**
 * 将一个新的变量声明添加到现有的变量声明列表中。
106 *
 * @param list 现有的变量声明列表。
108 * @param var 新的变量声明。
 * @return 返回更新后的变量声明列表。
110 */
struct var_decl_list *append_to_var_decl_list(struct var_decl_list *list, struct var_decl var) {
112     if (list == NULL) {
        return create_var_decl_list(var);
114     }
    struct var_decl_list *current = list;
116     while (current->next != NULL) {
        current = current->next;
118     }
    current->next = create_var_decl_list(var);
120     return list;
}
```