C-- 编译器实现

一、小组成员

- 3017218142 王昊宇
- 3017218140 王新阳
- 3017218063 刘兴宇
- 3017218158 李锐

二、使用说明

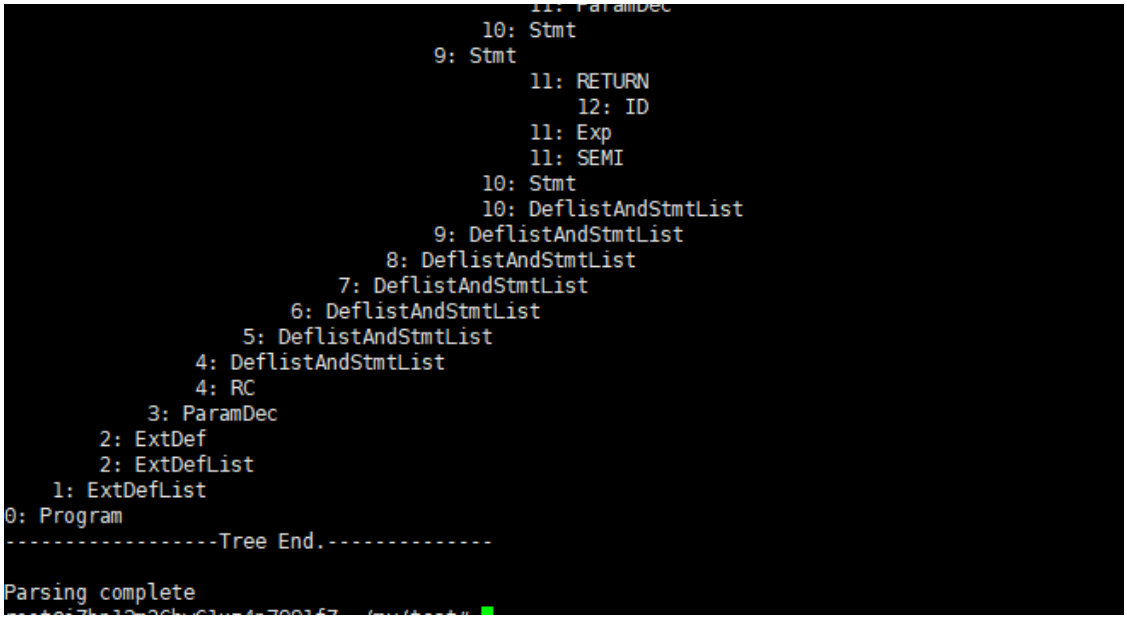
- 运行环境：Ubuntu 14.04 Ubuntu 16.04
- 本编译器所支持的词法和语法请参考第三第四小节
- 解压压缩包 运行命令
  unzip compiler.zip
- 进入文件夹
- 运行命令
  ./compiler test.cmm
  其中test.cmm可以替换成其他文件
  - 如果报错，则输出错误行号
  - 输出语法树
  - 产生语法树所用的产生式的推导/规约序列

```
root@iZbp13m26hw6luz4p7991fZ:~/my/test# ./compiler test.txt
FunDec -> MAIN LP RP
Specifier -> TYPE
VarDec -> ID
Exp -> INT
Dec -> VarDec ASSIGNOP Exp
DecList -> Dec
Def -> Specifier DecList SEMI
Specifier -> TYPE
VarDec -> ID
Exp -> INT
Dec -> VarDec ASSIGNOP Exp
DecList -> Dec
Def -> Specifier DecList SEMI
Specifier -> TYPE
VarDec -> ID
Exp -> INT
Dec -> VarDec ASSIGNOP Exp
DecList -> Dec
Def -> Specifier DecList SEMI
Exp -> ID
Exp -> ID
Exp -> MINUS Exp
Exp -> Exp ASSIGNOP Exp
Stmt -> Exp SEMI
Exp -> ID
Exp -> ID
Exp -> Exp RELOP Exp
Exp -> ID
Exp -> ID
Exp -> Exp ASSIGNOP Exp
Exp -> ID

Exp -> ID
Exp -> Exp PLUS Exp
Stmt -> Exp SEMI
DeflistAndStmtList -> EPSILON
DeflistAndStmtList -> Stmt DeflistAndStmtList
CompSt -> LC DeflistAndStmtList RC
Stmt -> CompSt
Stmt -> WHILE LP Exp RP Stmt
Exp -> ID
Stmt -> RETURN Exp SEMI
DeflistAndStmtList -> EPSILON
DeflistAndStmtList -> Stmt DeflistAndStmtList
DeflistAndStmtList -> Stmt DeflistAndStmtList
DeflistAndStmtList -> Stmt DeflistAndStmtList
DeflistAndStmtList -> Def DeflistAndStmtList
DeflistAndStmtList -> Def DeflistAndStmtList
DeflistAndStmtList -> Def DeflistAndStmtList
CompSt -> LC DeflistAndStmtList RC
ExtDef -> VOID FunDec CompSt
ExtDefList -> EPSILON
ExtDefList -> ExtDef ExtDefList
Program -> ExtDefList
Traverse Tree:
            3: void
                4: MAIN
                4: LP
                4: RP
            3: FunDec
                4: LC
                            7: TYPE
                        6: Specifier
                                    9: ID
                                8: VarDec
                                8: ASSIGNOP
                                    9: INT
```

...

```
                                                    15: Exp
                                                    15: PLUS
                                                        16: ID
                                                    15: Exp
                                                14: Exp
                                                14: SEMI
                                            13: Stmt
                                            13: DeflistAndStmtList
                                        12: DeflistAndStmtList
                                        12: RC
                                    11: ParamDec
```



三、词法分析

1. 概述

   词法分析器的作用是读取源程序生成词法单元，并过滤掉注释和空白。项目中的词法分析使用了 lex 。

2. 词元类型说明

   - INT

     INT表示的是整型常数。一个十进制整数由0~9十个数字组成，数字与数字中间没有空格之类的分隔符。

     除0之外，十进制整数的首位数字不为0。

     八进制或十六进制的形式。八进制整数由0-7八个数字组成并以数字0开头。

     十六进制整数由0-9、a-f十六个数字组成并以0x开头。

   - FLOAT

     FLOAT表示的是浮点型常数。一个浮点数由一串数字与一个小数点组成，小数点的前后必须有数字出现。

     浮点型常数还可以以指数形式表示。指数形式的浮点数必须包括基数、指数符号和指数三个部分，且三部分依次出现。基数部分由一串数字（0~9）和一个小数点组成，小数点可以出现在数字串的任何位置；指数符号为E或e；指数部分由可带-或者不带的一串数字组成，-必须出现在数字串之前。

   - ID
     ID表示的是标识符。标识符由大小写字母、数字以及下划线组成，但必须以字母或者下划线开头。
   - SEMI → ;
   - ASSIGNOP → =
   - RELOP → = | >= | == | <=
   - PLUS → +
   - MINUS → -
   - STAR → *
   - DIV → /
   - LP→(

- RP→)
- LC→{
- RC→}
- WHILE→while
- ELSE→else
- IF → if
- RETURN → return
- TYPE → int | float
- VOID→ void
- MAIN → main
- EASY→ continue|break
- FOR → for

四、语法分析

1. 概述

   语法分析接收词法分析器提供的记号串，检查记号串是否能由 c-- 规定的文法产生，并提示错误信息。

2. 语法规则

Program → ExtDefList

> 一段程序可以看作是定义的集合

ExtDefList → ExtDef ExtDefList

| ϵ

> 描述的是定义的集合

ExtDef → Specifier Dec SEMI

| Specifier FunDec CompSt

| VOID FunDec CompSt

> 定义：全局变量的定义、有返回值函数的定义、无返回值函数的定义

Specifier → TYPE

> int或者float

VarDec → ID

> 变量的声明，即标识符

FunDec → ID LP VarList RP　　//含参函数头的定义 func(a)

| ID LP RP　　//不含参函数头定义. func()

| MAIN LP RP

> 函数的声明：含参函数的声明，不含参数函数的声明，main函数的声明

VarList → ParamDec

> 函数参数列表，这里只支持一个参数的参数列表

ParamDec → Specifier VarDec

> 参数列表中，每个变量的声明例如int a

CompSt → LC DefListAndStmtList RC

> 函数体，由花括号包裹的定义和statement列表

DefListAndStmtList → Def DefListAndStmtList

| Stmt DefListAndStmtList

| ϵ

> declaration和statement的集合

Stmt → Exp SEMI

|CompSt

|RETURN Exp SEMI

|IF LP Exp RP Stmt

|IF LP Exp RP Stmt ELSE Stmt

|WHILE LP Exp RP Stmt

| FOR LP Exp SEMI Exp SEMI Exp RP Stmt

| FOR LP SEMI Exp SEMI Exp RP Stmt

| FOR LP Exp SEMI SEMI Exp RP Stmt

| FOR LP Exp SEMI Exp SEMI RP Stmt

| FOR LP SEMI Exp SEMI RP Stmt

| FOR LP SEMI SEMI RP Stmt

|EASY SEMI

> statement的定义：函数体，return语句，if语句，while循环，for循环，break语句，continue语句等

Def → Specifier Dec SEMI

> 局部变量的声明

Dec→VarDec

|VarDec ASSIGNOP Exp

> 局部变量声明的写法：形如a 或者 a=1

Exp → Exp ASSIGNOP Exp

| Exp RELOP Exp

| Exp PLUS Exp

| Exp MINUS Exp

| Exp STAR Exp

| Exp DIV Exp

|MINUS Exp

|ID LP Exp RP

|ID LP RP

| ID

| INT

| FLOAT

> 赋值语句，布尔表达式，加减乘除语句，取相反数，函数使用，标示符等语句

与C语言的区别：

- 程序不必满足有且只有一个main函数的条件
- 函数没有必要先声明再使用
- for 循环的第一个表达式，不支持声明局部变量

3. 语法树输出实现
    - 使用一个多叉树和一个单项链表构造并输出语法树
    - 多叉树的实现：多叉树的主要作用是构造各个子语法树（以非终结符为单位）
        - type：节点类型。type=0，代表为中间节点，type=1，代表叶子节点
        - data：节点值（终结符或非终结符）
        - length：子节点个数
        - leaves是一个节点指针的数组，长度固定为9，其中有效值个数为length个，存储指向子节点的指针

```
//树节点
typedef struct TreeNode{
    int type;
    int length;
    struct TreeNode* leaves[9];
    char* data;
}TreeNode, *Tree;
```

- 多叉树操作
  - 对于终结符，创建一个叶子节点，参数为终结符的值。默认type=1 ， length=0，叶子节点指针均指向空

```
//创建叶子节点
Tree createLeaf(char* rootData){
    int i = 0;
    Tree T = (Tree)malloc(sizeof(TreeNode));
    if (T != NULL){
        T -> type = 1;
        T -> data = rootData;
        T -> length = 0;
        for (i = 0 ; i < 9 ; i++){
            T -> leaves[i] = NULL;
        }
    }else{
        exit(-1);
    }
    return T;
}
```

  - 对于非终结符，创建一个中间节点，并连接对应的子节点。参数为非终结符值、子节点指针数组、子节点个数，type = 0

```
//链接叶子节点和根节点
Tree createTree(char* root, TreeNode** leaves, int length){
    int i = 0;
    Tree T = (Tree)malloc(sizeof(TreeNode));
    if (T == NULL){
        exit(-1);
    }else{
        T -> type = 0;
        T -> data = root;
        T -> length = length;
        for (i = 0 ; i < 9 ; i++){
            T -> leaves[i] = leaves[i];
        }
    }
}
```

- 链表实现：链表的主要作用是按照语法分析（自底向上）顺序存储终结符子语法树，用以连接各子语法树
  - data：指向一棵子语法树根节点的指针
  - next：指向链表中下一个节点

```
//链表节点
typedef struct ListNode{
    struct TreeNode* data;
    struct ListNode* next;
}ListNode, *LinkList;
```

- 链表操作：
  - 创建链表，返回头结点（data为空，next指向第一个节点）

```
//创建链表头节点（指向第一个节点）
LinkList createLinkList(){
    ListNode* L = (ListNode*)malloc(sizeof(ListNode));
    if ( L == NULL ){
        exit(-1);
    }
    L -> next = NULL;
    return L;
}
```

  - 头插入法，返回插入后的链表头结点

```
//链表头插入节点
LinkList linkListInset(LinkList L, TreeNode* newNode){
    ListNode* temp = (ListNode*)malloc(sizeof(ListNode));
    if ( temp == NULL ){
        exit(-1);
    }
    temp -> data = newNode;
    temp -> next = L -> next;
    L -> next = temp;
    return L;
}
```

- 构造语法树
  - 声明构造过程中使用到的变量

```
LinkList list = NULL; //链表头结点
LinkList head = NULL; //链表第一个节点
Tree T; //子语法树根节点
TreeNode* child[9]; //子节点指针数组
```

  - 在main函数中对其进行初始化

```
list = createLinkList();
T = NULL;
for (i = 0 ; i < 9 ; i++){
    child[i] = NULL;
}
```

  - 对于所有产生式，添加以下语义

```
ExtDef: Specifier ExtDecList SEMI
{
    printf("ExtDef -> Specifier ExtDefList SEMI\n"); //输出产生式
    head = list -> next;    //初始化head
    alloc(child);           //为子节点指针数组分配空间
    child[0] = (head->next) -> data;    //对于产生式右端非终结符，将链表中存储的子语法树根节点指针赋予它，先出现的非终结符在链表的后面
    child[1] = head -> data;
    child[2] = createLeaf("SEMI");  //对于产生式右端终结符，创建叶子节点
    T = createTree("ExtDef", child, 3); //连接子语法树
    list -> next = (head -> next) -> next;  //将本产生式右端非终结符对应的子语法树从链表中移除
    linkListInset(list, T); //插入新构造的子语法树
}
```

其中：

```
//为child分配内存空间
void alloc(TreeNode** child){
    int i;
    for (i = 0 ; i < 9 ; i++){
        child[i] = (TreeNode*)malloc(sizeof(TreeNode));
        if(child[i] == NULL){
            exit(-1);
        }
    }
}
```

- 输出语法树
  - 定义全局变量level，存储当前树的层数（从0开始）
  - 后序遍历语法树

```
//后序遍历
void traverseTree(Tree T, int level){
    int i = 0;
    if (T == NULL){
        return;
    }else{
        if (T -> type == 1){    //叶子节点
            if (T -> data != ""){   //epsilon不输出
                for ( i = 0 ; i < level ; i++){ //根据层数，输出4*层数个空格
                    printf("%-4s", " ");
                }
                printf("%d: %s\n", level, T -> data);   //输出"层数：值"
            }
        }else{  //中间节点
            for (i = 0 ; i < T -> length ; i++){    //递归遍历所有子节点，层数+1
                traverseTree(T -> leaves[i], level + 1);
            }
            //再变量中间节点
            if (T -> data != ""){
                for ( i = 0 ; i < level ; i++){
                    printf("%-4s", " ");
```

```
                }
                printf("%d: %s\n", level, T -> data);
            }
        }
    }
}
```

- 对初始符号产生式，在构造语法树的语义后，添加以下语义

```
head = list -> next; //重新初始化head
T = head -> data;   //T为当前开始符号子语法树根节点，即语法树根节点
printf("Traverse Tree: \n");
level = 0;
traverseTree(T, level); //以根节点为0层，遍历
printf("------------------Tree End.--------------\n");
```

五、项目展望

本编译器基本完成了题目中的要求，但是也存在以下不足：

首先项目中的词法语法满足了是C语言的一个子集，但是缺少了很多作为编程语言中必不可少的约束。

此外，本节课程主要教授的是编译器前端部分，而我们并没有使用上课讲过的中间语言表达形式进行输出。这些都是在项目中存在的不足。

课程虽然已经结束，但是学习的脚步不会停止，我们将不断的完善这个项目。