词法分析与语法分析

程序功能(必做+选做1.1)

数组访问错误处理

对于实验说明中，

> 输入文件的同一行中保证不出现多个错误，你的程序需要将这些错误全部报告出来，每一条错误提示信息在输出中单独占一行。

程序对数组访问错误的处理方式和讲义中稍有不同。

```
int main()
{
    float a[10][2];
    int i;
    a[5,3] = 1.5;
    if (a[1][2] == 0) i = 1 else i = 0;
}
```

程序将报告三个错误，其中第4行有两个错误，一个是“]”缺失，一个是语句不合理。

```
Error type B at Line 4: Missing "]".
Error type B at Line 4: Unvalid statement.
Error type B at Line 5: Missing ";".
```

这是为错误恢复提供便利，如果单纯将错误认定为“]”缺失，程序将无法处理如下输入文件，因为不能确保任何“]”缺失都可以有相应的恢复策略。

```
int main()
{
    int a[10];
    int b[10];
    int c;
    c = b[a[1];
}
```

输出：

```
Error type B at Line 6: Missing "]".
```

C--语言和C语言的文法差异处理

对于C--语言中没有的标识符 `for`，`double` 等进行了恢复处理，对输入文件中类似的标识符进行识别，如下，

```
int main()
{
    int float = 1;
    float 1err = 1.0;
}
```

输出：

```
Error type B at Line 3: Unvalid name of variable.
Error type B at Line 4: Unvalid identifier.
```

实现思路

文件组织

```
Code目录下：
.
├── Makefile
├── lexical.l
├── main.c
├── parse.c
├── parse.h
└── syntax.y
```

数据结构

定义 `TreeNode` 类型结构体，建立语法多叉树。

```
typedef struct TreeNode {
    char name[32];  //词法单元
    char text[32];  //词素
    int lineno;     //行号
    int childsum;   //子节点数目
    struct TreeNode* child[MAX_CHILD_NUM];
} Node;
```

主要功能函数如下，

```
Node* createNode(char*, char*);
void insertNode(int, Node*, ...);
void printNode(char*, char*);
void printTree(Node *, int);
```

实现过程

1. **检验词法分析程序** 根据C--文法构造相应的正则表达式，并额外定义了常见错误形式的正则表达式(主要针对C--和C语言的文法差异)，充分利用了正则表达式的匹配规则，权衡了正确性了代码的可读性，如下，

    - 不合法的十进制
    - 不合法的八进制
    - 不合法的十六进制
    - 不合法的单精度浮点数
    - +=，-=，/*，*/，//

2. **添加文法的产生式** 根据C--语言文法添加产生式，并通过 `%left` 、 `%right` 和 `%nonassoc` 以及它们的顺序对终运算符的优先级(precedence)以及结合性(associativity) 进行规定。

3. **恢复解决文法歧义** 悬空else问题的解决方案参照讲义中增加 `LOWER_THAN_ELSE` 算符完成，其他二义文法均使用词法单元error进行错误恢复处理。

4. **构建并打印语法树** 先序遍历多叉树，按照讲义要求递归打印语法树节点。

代码风格

- 把功能的底层实现封装到函数中，增加代码的可读性。
- 开启宏定义 `#define PRINT_DEBUG` 进入调试模式，可以额外打印结点的词法单元类型，并在语义恢复之后输出语法树(即使存在语法错误)。

写在最后

和室友聊天说起编译原理实验的讲义，我们都觉得相对PA和OS，这本指导手册的可读性更高一些。 坦白的说，之前的两个实验使我对稍具规模的实验任务产生了自然而然的畏惧感。 不过欣喜的是，我感受到了实验任务和理论课程的直接联系非常紧密，这使我在完成实验一的过程中有不小的成就感。

最后非常感谢助教学长和许畅老师对我的问题详尽的解答!

语义分析

程序功能(必做+选做2.3)

实现思路

数据结构

符号表的设计

符号表使用Hash表，将基本类型、数组类型、结构类型、函数类型变量存储在一个统一的符号表中，提高插入和查找的效率。

符号表：

```
struct SymbolTable {
    char name[32];
    Type type;
    int lineno;
    bool occupied;
} symboltable[0x4000];
```

变量类型：

```
typedef struct Type_
{
    enum { BASIC, ARRAY, STRUCTURE, FUNCTION } kind;
    union
    {
        int basic;
        struct { Type elem; int size; } array;
        FieldList structure;
        Function function;
    } u;
} Type_;
```

其中，枚举类型变量 `BASIC` ， `ARRAY` ， `STRUCTURE` ， `FUNCTION` 变量类型，在 `getNodeType` 和 `handleExp` 中作为返回类型的标识。实现假设每种类型的变量个数不会超过10个，因此使用了 `类型 * 10 + 序号` 作为标识。

函数类型：

```
typedef struct Function
{
    enum { INT, FLOAT } returnvalue;
    int argsum;
    int argbasic[8];
} Function;
```

结构体类型：

```
typedef struct FieldList_
{
    char name[32];
    Type type;
    FieldList tail;
} FieldList_;
```

结构体的管理

使用链表结构存储结构体，包括结构体名称和其中的成员变量。

```
typedef struct StructNode_ {
    int no;
    char name[32];
    FieldList children;
    StructNode next;
} StructNode_;
```

文法属性的处理

重新遍历实验一中建立的树，对文件进行语义分析。使用地址传递函数参数和返回值相结合的方式对文法的继承属性和综合属性进行处理。

```
// ExtDefList -> ExtDef ExtDefList | ϵ
void handleExtDefList(Node* root) {
    if (Childsum == 0) return;
    // Iteration is better than recursion.
    while (root != NULL) {
        handleExtDef(Child(0));
        root = Child(1);
    }

    #ifdef HANDLE_DEBUG
    printf("handleExtDefList bingo\n");
    #endif
}
```

全局变量的使用

由于函数传递参数的缺陷，使用全局变量进行相关辅助。

- `PARA` 用于标识所在位置是否属于函数的参数定义。
- `IN_STRUCT` 用于标识结构体嵌套的深度。
- `FLAG` 用于标识 `ExtDef` 的产生式的类型。

实现过程

1. **设计数据结构，规划整体实现流程**
2. **将变量加入符号表，处理相关语义错误**
3. **进行变量声明和类型检查**
4. **构造测试用例进行调试，尽量补充遗漏的特例**

代码风格

优点：

- 把功能的底层实现封装到函数中，增加代码的可读性。
- 在实验一的基础上增加宏定义 `#define PHASE_SEM` 进入调试模式，可以打印插入符号表的相关信息。（将实验一中的宏 `PRINT_DEBUG` 更名为 `PRINT_TRACE` 。）
- 使用多个参数的宏定义，子孙节点的访问和控制，最多支持6个参数。

有待提高：

- 对指针的操作不是很熟练，相关代码实现结构性相对较差，给实现和阅读带来不便。
- 对于结构体嵌套和多维数组以及一些局部变量的处理存在硬编码。

写在最后

报告至此，我的心情十分复杂。实现编译器的语义分析的复杂度远超出我之前的想象。体会最深的一点是有必要在动手之前设计出合理的数据结构，不然将会带来极大的困难。设计模式是否友好奠定了整个工程的基础。大学三年，第一次在写实验的过程中感受如此深刻。多次的代码重构使我有点烦躁，但不得不承认，加深了我对理论知识的理解，提高了自己的动手能力，感觉收获很大。

中间代码生成

程序功能(必做+部分选做)

程序接口说明

```
./parser in out
```

其中 `in` 为输入文件，转换生成的中间代码将存储在 `out` 文件中。输出目标文件缺省值为 `out.ir` ，另外还会生成 `original.ir` 文件，写入内容为未经过优化的代码。

中间代码生成

将中间代码输出成**线性结构**，存储在**双向链表**的结构中。

中间代码优化

实现了**常量折叠**和**代数运算简化**的优化功能。具体说来，先定位将常量赋值给临时变量的赋值语句，并在其后使用这个临时变量时将其替换为对应的常量；再对可能会由此产生一些可以直接进行计算的四则运算表达式进行求值。之后**迭代**进行常量赋值表达式消除和代数运算简化，直到产生的中间代码条数不发生变化，具体实现在 `optimize.c` 文件中的 `interCodeOptimize()` 函数。

实现思路

相关源码文件

```
.
├── ir.c
├── translate.c
└── optimize.c
```

数据结构设计

对于每一条中间代码，使用 `InterCode` 结构体类型，设计 `Operand_` 结构体类型表示每一个对应操作的相关信息。定义在 `ir.h` 文件中。

```
typedef struct Operand_ {
    enum { OP_VARIABLE, OP_TEMPORARY, OP_CONSTANT, OP_ADDRESS, OP_LABEL, OP_FUNCTI
ON, OP_VALUE, OP_SIZE } kind;
    union {
        int no;
        int value;
    } u;
} Operand_;

typedef struct InterCode {
    enum {
      IR_ADD, IR_SUB, IR_MUL, IR_DIV, IR_IFGOTO,
      IR_LABEL, IR_FUNC, IR_GOTO, IR_RET, IR_DEC, IR_ARG, IR_PARAM, IR_READ, IR_WR
ITE,
      IR_ASSIGN, IR_GETADD, IR_GETVAL, IR_SETVAL, IR_CALL
    } kind;

    union {
        struct { Operand x; } op;
        struct { Operand x, y; } biop;
        struct { Operand x, y, z; } triop;
        struct { Operand x, y, z; Relop relop; } ifop;
    };
} InterCode;
```

使用双向链表的结构存储中间代码，具体实现参考了Linux内核的 `list_head` 结构。

```
typedef struct InterCodes
{
    InterCode intercode;
    struct list_head list;
    bool isLeader;
} InterCodes;
```

翻译流程的架构设计

类似实验二中的自顶向下分析方法，从 `Program` 开始进行逐层函数调用，在底层进行中间代码生成，必要的时候返回调用者 `place` 参数，具体功能实现参照翻译模式。临时变量和标号的计数使用了全局变量。对于变量和函数的符号表，原则上可以沿用语义分析中的设计，我对之前的版本进行了完善。

数组的翻译模式设计

为提高编译器运行效率，使用循环而非递归进行数组的翻译模式设计，具体实现在 `translate.c` 中的 `translate_Array` 。

以四维数组 `int array[7][8][9][10]` 为例说明设计的翻译模式，设 `&v1` 表示 `array` 的地

址，`v2`，`v3`，`v4`，`v5`分别对应`i`，`j`，`k`，`s`的值。计算数组元素`array[i][j][k][s]`的地址，有

```
&v1 + v2 * 8 * 9 * 10 * 4 + v3 * 9 * 10 * 4 + v4 * 10 * 4 + v5 * 4
= &v1 + v2 * 8 * 9 * 10 * 4 + v3 * 9 * 10 * 4 + v4 * 10 * 4 + v5 * 4
= &v1 + (((v2 * 8 + v3) * 9 + v4) * 10 + v5) * 4
```

对应的中间代码翻译为

```
t2 := #8
t3 := #9
t4 := #10

t5 := v2 * t2
t6 := t5 * t3
t7 := t6 * t4
t8 := t7 * #4
t15 := &v1 + t8

t9 := v3 * t3
t10 := t9 * t4
t11 := t10 * #4
t16 := t15 + t11

t12 := v4 * t4
t13 := t12 * #4
t17 := t16 + t13

t14 := v5 * #4
t18 := t17 + t14
```

写在最后

个人觉得中间代码生成部分的必做实验难度要略低于语义分析部分。按照讲义上的指导一步步完成相对比较顺利。其中，耗时最长的数组的翻译模式设计，我使用了从特殊到一般的归纳法。先写出低维数组的翻译函数，然后类比推理出任意维的数组的翻译函数。另外，对于中间代码的优化，虽然我基本理解了课上老师介绍的使用DAG进行基本块内部优化的理论，但是在实践过程中遇到了重重的困难。经过两天的尝试，为了尽量保证中间生成代码的正确性，我最后选择了相对保守的、实现复杂度较低的优化策略，在实现复杂度、优化效果和正确性之前进行了权衡。