编译原理Lab3实验报告

201220113 贺保定

运行方式

使用Code目录下的Makefile进行构建。在Code目录下运行“make parser”，会在Code生成名为Parser的可执行文件。然后使用“./parser 输入文件 输出文件”的方式执行中间代码生成。

功能介绍

本次实验实现了所有必做任务，在实验给出的假设下，对包含顺序语句，分支控制，函数调用等基本情况的cmm源文件，生成能够产生预期结果的中间代码。根据分配的任务号，另外完成了选做任务二，实现了数组相关的语言功能，支持一维及以上数组的创建和访问，一维数组作为参数，一维数组直接拷贝赋值。

实现细节

文件结构

为了完成本次实验的要求，新增文件ir.h、ir.c，存放中间代码相关的函数定义；新增translate.h、translate.c，存放翻译中间代码的逻辑。

全局符号表

由于本次实验保证无全局变量、变量均具有全局作用域，而上次本小组分配到的任务是实现局部变量，因此，稳妥起见，我重写了具有全局作用域的符号表，并在本次实验中使用。

```
struct GlobalSymbol {
    char* name;
    int alias;
    struct PlainType* type;
    struct GlobalSymbol* next;
};
```

全局的符号定义如上。符号除了名字外，根据出现顺序，为其分配一个序号作为别名。全体符号存放在链表中。可以调用`insert_symbol(char *name)`函数为符号表插入新的符号。如果传入的名称为`NULL`，则默认命名为`t1、t2···`的形式。

中间代码

每一条中间代码使用`IR`结构体表示。全部中间代码使用双向循环链表组织，用`ir_head`和`ir_tail`指示头部和尾部。使用`insert_ir(struct IR* ir)`向链表尾部插入新的中间代码。

翻译流程

生成中间代码的过程，是对语法树进行深度优先搜索的过程（几乎所有地方）。在`translate.h`中，使用形如`translate_TOKEN`的方式命名这些函数。以下对几种语法单元的翻译方法进行介绍。

- 函数定义

```
FUNDEC -> ID LP VarList RP
        | ID LP RP
```

搜索到函数名`ID`时，加入中间代码`FUNCTION ID`。然后搜索`VarList`。搜索每个`ParamDec`，从`Specifier`获取类型信息。最终翻译`VarDec`。`VarDec`可能是基本类型的变量，也可能是数组类型的变量。这里获取参数类型的翻译方法和函数内部的变量声明类似。最终得知形式参数的名称`ID`后，将`Prama ID`加入中间代码列表。

- 基本变量声明

```
VarDec → ID
        | VarDec LB INT RB
```

基本变量仅具有简单的类型信息。对于每个变量的类型信息，都初始化为`BASIC`，并且具有`int`类型，鉴于假设无`float`型变量。

```
void translate_DecList(TreeNode *root, struct PlainType* plain_type) {
    struct PlainType* new_type = new(PlainType);
    new_type->info.basic = plain_type->info.basic;
    new_type->info.array.width = NULL;
    ...
}
```

- 数组变量声明

如果`VarDec`产生多于一个的`TOKEN`，说明此时正在声明一个数组。此时需要将类型由`BASIC`更改为`ARRAY`。另外，还需要维护数组的宽度信息。得到产生式中的`INT`的值后，将此值存放入类型结构体中维护数组宽度信息的链表中。最后，得到`ID`后，通过宽度链表计算数组占用的总空间。通常而言，此时应当向`IR`列表中添加`DEC ID SIZE`语句，为数组创建空间。但是，如果该数组声明出现在函数参数中，则不应该创建空间，因为此处实际上仅仅传入数组的首地址，数组的空间在之前已经创建完毕。因此，`translate.c`中设有全局变量`defining_param`，用以标志当前是否在翻译函数参数，从而决定是否需要声明数组的地址空间。

- 控制流语句

控制流语句被翻译为`LABEL`、`GOTO`、`IF GOTO`和普通语句的组合。这里着重叙述条件语句的翻译。

由于条件表达式并未被保证仅由条件运算符组成，所以，需要更一般地处理条件表达式。具体而言，对于

```
    IF LP Exp RP Stmt
```

此处的Exp会直接使用`translate_Exp`进行翻译，以确保各种运算都被兼容。不过，中间代码中并没有关系运算`> < >= <= == !=`和逻辑运算`&& || !`。所以，对于关系运算，将两侧表达式的结果相减，再利用`IF a RELOP b GOTO label`进入合适的赋值语句，将此表达式的结果赋为1或0。对于`&& ||`，同样利用有条件跳转实现运算和短路求值特点。

- 数组访问

```
Exp -> Exp LB Exp RB
    | ID
```

首先翻译第一个Exp，获得基址。然后翻译第二个Exp，获得偏移量。最后将基址加偏移量的结果存放入`place`变量中。翻译过程中需要维护类型信息。每有一重数组访问，就将数组的维度降低一维。例如，对于`int a[3][4]`，原先的宽度链表为`3 -> 4 -> null`，语句`a[2]`的值的宽度链表为`4 -> null`。将访问结果解释为一个子数组有利于处理赋值语句。

- 赋值

对于赋值语句的右方，创建一个临时变量承接右侧表达式的结果。右侧表达式计算得到的可能是一个数，也可能是一个地址，如`a[1][2]`。此时查看等号左侧。若左侧是基本变量，由于语义保证正确，取出右侧计算得到的地址存储的值，赋予左侧。若左侧同样为数组访存，则按照类型信息限制的宽度，顺次将左侧基址开始的位置赋予右侧相应位置上的值。即使左右实际上不是子数组，而都只具有宽度4，例如对于二维数组`a`、`b`，出现语句`a[1][2] = b[3][5]`。对这样的语句，前述的处理方式也完全正确，因为根据数组类型信息计算得到的值会表明左右两个地址的宽度均为1。

- 函数调用

  此处主要的麻烦在于处理实参列表。同样使用链表倒序保存参数，最后为每个参数加入`ARG t`式的中间代码。需要注意如果当前在处理`write`函数，则不需要输出传参语句。这同样使用全局的标志作决断。

- 表达式

  需要注意，取负语句`MINUS a`应当译为`#0 - a`。