编译原理实验报告

------2020200982 闫世杰

实验目标：

设计并实现一个基于 Sysy 语言的编译器

实现过程：

1. 词法分析：基于 flex 实现词法分析

由于后续语法分析的需要，绝大多数内容都需要进行精准匹配，需要用到正则表达式的地方只有数字和标识符，因此该过程较为简单

数字及标识符正则式如下，其余均为精确匹配，此处省去：

```
OCTINT 0[1-7][0-7]*
DECINT [1-9][0-9]*|0
HEXINT 0[Xx][1-9a-fA-F][0-9a-fA-F]*
IDENTIFIER [a-zA-Z_][a-zA-Z0-9_]*
```

需要注意的地方是对注释的匹配，个人是利用 flex 的状态转换功能实现

对于单行注释，直接使用正则匹配即可
对于块注释，由于可以跨越多行，因此使用状态转换来实现，匹配到/*时进入注释态，任何匹配都不会进行，匹配到*/时恢复到正常态，继续进行匹配

```
LINECOMMENT "//".*\n
%x BLOCKCOMMENT
%%
"/*" {BEGIN BLOCKCOMMENT;}
<BLOCKCOMMENT>. {}
<BLOCKCOMMENT>\n {lineno++;}
<BLOCKCOMMENT>"*/" {BEGIN 0;}
{LINECOMMENT}  {lineno++;}
```

对于词法分析所匹配到的所有词都将作为一个终结符传递给语法分析器，进行后续的工作

2. 语法分析：基于 bison(yacc)对匹配的词组进行语法分析

只需要写出 Sysy 的语法规则，并给出相关的语义动作.
具体的语法规则在文档中都已经给出了，只需要稍加改进，实现其中包含的正则匹配式，同时定义相关的优先级或者改写文法来消除冲突即可

正则匹配式的实现：
1. [T]:T 可有可无的情况：
把包含[T]的语句写成两句(含 T\不含 T)
2. {T}:T 可以 0 次或多次出现：
改写成 Ts->Ts T|empty 即可

冲突的消除：
1. 四则运算优先级：
给出的语言定义文法已经解决了该问题
2. if_else 移进归约冲：
利用 yacc 的优先级定义来解决

```
%nonassoc IFF
%nonassoc ELSE
| IF LPAREN Cond RPAREN Stmt ELSE Stmt {
}
| IF LPAREN Cond RPAREN Stmt  %prec IFF{
}
```

具体的语法规则实现在 parser.y 中，在此不多叙述

由于是先利用语法分析的过程生成语法树，然后分析整个语法树来完成语义分析并生成汇编代码，因此在整个语法分析中，任务不是特别多，只进行语法树节点的创建，语法树中父子关系的指向

记录变量的结构体

```
/*--------------------struct type of var----------------*/
struct Type {
    bool constvar;
    ValueType type;
    Type(ValueType valueType){ //创建变量类型值
    };
    void copy(Type* a){ //复制类型值
    };
    unsigned short paramNum; //如果变量是函数,记录参数个数
    Type* paramType[MAX_PARAM]; //参数列表
    Type* retType; //返回值类型
    unsigned int dim;  // 如果变量是数组,记录维度
    int dimSize[MAX_ARRAY_DIM]; //记录维度值
    unsigned int visitDim = 0; //记录正在访问的维度
    int getSize(){ //返回变量的大小
    };
};
```

记录语法树节点的结构体

```
/*---------------struct tree node-------------------*/
struct TreeNode {
    int lineno;
    TreeNode* child = nullptr;
    TreeNode* sibling = nullptr; //采用child_link方式记录孩子,方便后续语义分析

    NType nodeType; //记录节点类型: 变量/常量/运算符/表达式/函数参数......
    OType optype; //如果节点是运算符, 记录运算符类型
    StmtType stype;//如果节点是表达式状态, 表达式类型
    Type* type;//如果节点是变量/常量, 记录变量类型:数字/数组/函数
    int int_val; //如果节点是int量, 记录值
    bool b_val; //如果节点是bool量, 记录值
    string var_name; //节点名称
    string var_scope; //变量作用域标识符

    TreeNode(int lineno, NType type); //创建建立值节点
    TreeNode(TreeNode* node);   //节点的复制
    void Relate(TreeNode*); //增加第一个孩子节点
    void RelateSib(TreeNode*); //增加孩子节点
    int getVal(); //如果节点是int/bool,返回节点的值
    int childnum = 0; //孩子节点个数
    void findReturn(vector<TreeNode *> &retList); //找到return所在节点,将其添加到retlist中
    /* -------------asm------------ */
    Label label; //记录label, if_else while bool语句使用
    void gen_var_decl(); //检查并生成变量的汇编代码
    void genCode(); //生成汇编代码函数入口
    string new_label(); //创建新的label
    void get_label(); //为if_else while bool语句分配label
    string getVarNameCode(TreeNode* p); //生成取值操作的汇编代码
};
```

语法分析之后会生成对应的语法树,语法树中记录着整个程序的信息,后续的语义分析会根据语法树,生成对应的汇编代码

3.语义分析：遍历分析语法树,生成最终的汇编代码

从根节点开始,遍历整棵语法树,根据语法树的节点中记录的节点的类型来进行具体汇编代码的输出

根节点:即程序的开始节点
输出整个程序所定义的全局变量,并递归进行后续函数及语句的代码生成

```
case NODE_PROG:
    gen_var_decl();
    cout << "\t.text" << endl;
    while (p) {
        if (p->nodeType == NODE_STMT && p->stype == STMT_FUNCDECL)
            p->genCode();
        p = p->sibling;
    }
    break;
```

函数调用:传入参数并进行 call 操作

```
case NODE_FUNCALL:
    p->sibling->child->genCode();
    cout << "\tpushq\t%rax" << endl;
    pSize += this->child->type->paramType[0]->getSize(); //传入参数,只支持一个参数
    /*---------call, scanf printf特殊处理--------*/
    if(child->var_name == string("printf"))
        cout << "\tcall\t" << "printf@PLT" << endl << "\taddq\t$" << pSize << ",%rsp" << endl;
    else if (child->var_name == string("scanf") )
        cout << "\tcall\t" << "__isoc99_scanf@PLT" << endl << "\taddq\t$" << pSize << ",%rsp" << endl;
    else
        cout << "\tcall\t" << child->var_name << endl << "\taddq\t$" << pSize << ", %rsp" << endl;
    break;
```

函数声明语句:分配内存,label,并递归进行函数内部汇编代码生成

```
case STMT_FUNCDECL:
    cycleStackTop = -1;
    pFunction = this;
    get_label();
    cout << "\t.globl\t" << p->sibling->var_name << endl
         << "\t.type\t" << p->sibling->var_name << ", @function" << endl
         << p->sibling->var_name << ":" << endl;
    gen_var_decl();
    cout << "\tpushq\t%rbp" << endl
         << "\tmovq\t%rsp, %rbp" << endl;
    cout << "\tsubq\t$" << -stackSize << ", %rsp" << endl; // 在栈上分配局部变量
    p->sibling->sibling->sibling->genCode(); // 内部代码递归生成
    cout << this->label.next_label << ":" << endl; // 产生返回标签代码
    cout << "\taddq\t$" << -stackSize << ", %rsp" << endl; // 清理局部变量栈空间
    cout << "\tpopq\t%rbp" << endl
         << "\tret" << endl;
    pFunction = nullptr;
    break;
```

if_else while 语句,分配 label,设置入口及跳转

```
case STMT_IF:
    get_label();
    cout << label.begin_label << ":" << endl;
    this->child->genCode();
    cout << label.true_label << ":" << endl;
    this->child->sibling->genCode();
    cout << label.false_label << ":" << endl;
    break;
case STMT_IFELSE:
    get_label();
    cout << label.begin_label << ":" << endl;
    this->child->genCode();
    cout << label.true_label << ":" << endl;
    this->child->sibling->genCode();
    cout << "\tjmp\t" << label.next_label << endl;
    cout << label.false_label << ":" << endl;
    this->child->sibling->sibling->genCode();
    cout << label.next_label << ":" << endl;
    break;
case STMT_WHILE:
    get_label();
    cycleStack[++cycleStackTop] = this;
    cout << label.next_label << ":" << endl;
    this->child->genCode();
    cout << label.true_label << ":" << endl;
    this->child->sibling->genCode();
    cout << "\tjmp\t" << label.next_label << endl;
    cout << label.false_label << ":" << endl;
    cycleStackTop--;
    break;
```

由于分类较多,其余表达式的汇编生成过程不再一一列举,具体内容在 tree.cpp 中

label 的设置:按 ppt 上的方法,用共享 label 的方法来完成回填操作

让子语句与父语句共享一份 label,在子语句修改 label 是父语句也会自动完成修改

```
case STMT_FUNCDECL:
    this->label.begin_label = this->child->sibling->var_name;
    this->label.next_label = ".LRET_" + this->child->sibling->var_name;
    break;
case STMT_IF:
    this->label.begin_label = new_label();
    this->label.true_label = new_label();
    this->label.false_label = this->label.next_label = new_label();
    this->child->label.true_label = this->label.true_label;
    this->child->label.false_label = this->label.false_label;
    break;
case STMT_IFELSE:
    this->label.begin_label = new_label();
    this->label.true_label = new_label();
    this->label.false_label = new_label();
    this->label.next_label = new_label();
    this->child->label.true_label = this->label.true_label;
    this->child->label.false_label = this->label.false_label;
    break;
case STMT_WHILE:
    this->label.begin_label = this->label.next_label = new_label();
    this->label.true_label = new_label();
    this->label.false_label = new_label();
    this->child->label.true_label = this->label.true_label;
    this->child->label.false_label = this->label.false_label;
    break;
default:
    break;
}
```

if_else while 的 label 实现↑　　布尔表达式的 label 实现↓

```
case NODE_OP:
    switch (optype)
    {
    case OP_AND:
        child->label.true_label = new_label();
        child->sibling->label.true_label = label.true_label;
        child->label.false_label = child->sibling->label.false_label = label.false_label;
        break;
    case OP_OR:
        child->label.true_label = child->sibling->label.true_label = label.true_label;
        child->label.false_label = new_label();
        child->sibling->label.false_label = label.false_label;
        break;
    case OP_NOT:
        child->label.true_label = label.false_label;
        child->label.false_label = label.true_label;
        break;
    default:
        break;
    }
    break;
default:
    break;
}
```

由于是先生成语法树,再进行语法树分析生成汇编代码,因此整个过程比较繁琐
由于对函数调用的理解不是特别深刻,在进行参数传递以及内存分配的时候存在存在着一些问题,因此函数调用功能不能正确完成
由于语法树节点基本是重新实现,此次生成的语法树节点类型的划分更加精细,语法树节点包含的内容以及语法树的节点远多于上次的语法实验,因此没有保留上次语法实验的做图部分

程序的主入口(main) 在 parser.y 文件最下方的代码段
执行 make 指令,会生成名为 complier 的可执行文件
会出现下列报错,查阅发现是 gcc 误报,详情见链接文章:
https://lists.gnu.org/archive/html/help-bison/2021-01/msg00021.html

```
parser.cpp: In function ‘int yyparse()’:
parser.cpp:2249:18: warning: ‘void free(void*)’ called on unallocated object ‘yyssa’ [-Wfree-nonheap-object]
 2249 |     YYSTACK_FREE (yyss);
parser.cpp:1055:16: note: declared here
 1055 |     yy_state_t yyssa[YYINITDEPTH];
      |                ^~~~~
```

执行 make run
会在 data 目录下生成 4 个 .s 文件,即所需的汇编代码
使用 gcc -static name.s -o name(静态链接方式) 生成可执行文件

由于函数调用汇编生成的过程存在问题,设计函数调用的代码执行时都会发生段错误,不涉及函数调用的代码可正常执行,但不能判断正确性(个人查看汇编,感觉应该没什么问题)

汇编代码生成的参考地址:
https://github.com/MilkyBoat/compilor_experiment