Contents

1 序言

我们实现的是一个类 C 语言的编译器，该语言包含了 C 语言的绝大部分特性，该编译器可以将 C 语言的代码转化成 x86 汇编指令。

组员的分工如下：

- 赵竟霖：代码生成和优化
- 黄文璨：语义分析和错误处理
- 金连源：词法分析和语法分析

2 词法分析

2.1 简介

词法分析（英语：lexical analysis）是计算机科学中将字符序列转换为单词（Token）序列的过程。进行词法分析的程序或者函数叫作词法分析器（Lexical analyzer，简称 Lexer），也叫扫描器（Scanner）。在本实验里面读取的大都是和 C 语言代码中相关的 token，程序是用 ply.lex 实现的，它的底层原理是有限状态机。执行函数以后的返回结果是 token 的序列。

2.2 实现细节

2.2.1 C 语言关键字 (keyword)

在 ply.lex 下，C 语言的关键字必须单独的处理，不然会出现二义性的问题。以下是编译器中关键字以及它的 token 表达形式：

- 'if' : 'IF'
- 'else' : 'ELSE'
- 'while' : 'WHILE'
- 'return' : 'RETURN'
- 'break' : 'BREAK'
- 'void' : 'VOID'
- 'int' : 'INT'
- 'float' : 'FLOAT'
- 'char' : 'CHAR'
- 'double' : 'DOUBLE'
- 'struct' : 'STRUCT'
- 'for' : 'FOR'
- 'continue' : 'CONTINUE'
- 'putchar' : 'PUTCHAR'
- 'getchar' : 'GETCHAR'

2.2.2 C 语言其他符号

除了关键字以外，我们还需要处理 C 语言其他的符号，下面列出它们所表达的含义以及它们所对应的正则表达式：

token	解释	正则表达式
PLUS	加号	r'+'
MINUS	减号	r'-'
TIMES	乘号	r'*'
DIVIDE	除号	r'/'
DOUBLEPLUS	加加	r'++'
DOUBLEMINUS	减减	r'–'
PLUSEQUAL	加等于	r'+='
MINUSEQUAL	减等于	r'-='
TIMESEQUAL	乘等于	r'*='
DIVIDEEQUAL	除等于	r'/='
SINGLEEQUAL	赋值	r'='
DOUBLEEQUAL	等于判断	r'=='
DOUBLEPIPES	或运算	r'‖'
DOUBLEAMPERSAND	与运算	r'&&'
BANGEQUAL	不等于	r'!='
LPAREN	左圆括号	r'('
RPAREN	右圆括号	r')'
LCURLY	左大括号	r'{'
RCURLY	右大括号	r'}'
LSQUARE	左方括号	r'['
RSQUARE	右方括号	r']'
SEMICOLON	分号	r';'
COMMA	逗号	r','
ID	标识符	r'[a-zA-Z_][a-zA-Z_0-9]*"
LANGLE	小于符号	r'<'
RANGLE	大于符号	r'>'
LANGLEEQUAL	小于等于	r'<='
RANGLEEQUAL	大于等于	r'>='
EXCLAMATION	感叹号	r'!'
PERCENT	百分号	r'%'
DOT	点运算符	r'.'
ADDR	取地址符	r'&'
LEFTARROW	箭头符号	r'->'
FLOATLITERAL	浮点数值	r'\d+.\d+'
INTLITERAL	整数值	r'\d+'
STRINGLITERAL	字符串值	r'\ " (\ . | [\ \ "])*\""

2.3 测试结果

考虑如下 C 语言代码

```
int f;
int k;

int go(int* b, int a)
{
    int fk;
    double t;
    int g;
    if(a > 0){
        g = a * go(b, a-1);
    }else{
```

```
        g = 1;
    }
    *b = *b + g;
    k = k + g;
    return g;
}

int main(void)
{
    k = 0;
    f = go(&k, 5);
    printf("%d\n", f);
    printf("%d\n", k);
    return 0;
}
```

它的对应的 token 序列如下：

```
LexToken(INT,'int',2,1)
LexToken(ID,'f',2,5)
LexToken(SEMICOLON,';',2,6)
LexToken(INT,'int',3,8)
LexToken(ID,'k',3,12)
LexToken(SEMICOLON,';',3,13)
LexToken(INT,'int',5,16)
LexToken(ID,'go',5,20)
LexToken(LPAREN,'(',5,22)
LexToken(INT,'int',5,23)
LexToken(TIMES,'*',5,26)
LexToken(ID,'b',5,28)
LexToken(COMMA,',',5,29)
LexToken(INT,'int',5,31)
LexToken(ID,'a',5,35)
LexToken(RPAREN,')',5,36)
LexToken(LCURLY,'{',6,38)
LexToken(INT,'int',7,44)
LexToken(ID,'fk',7,48)
LexToken(SEMICOLON,';',7,50)
LexToken(DOUBLE,'double',8,56)
LexToken(ID,'t',8,63)
LexToken(SEMICOLON,';',8,64)
LexToken(INT,'int',9,70)
LexToken(ID,'g',9,74)
LexToken(SEMICOLON,';',9,75)
LexToken(IF,'if',10,81)
LexToken(LPAREN,'(',10,83)
LexToken(ID,'a',10,84)
LexToken(RANGLE,'>',10,86)
LexToken(INTLITERAL,0,10,88)
LexToken(RPAREN,')',10,89)
LexToken(LCURLY,'{',10,90)
LexToken(ID,'g',11,100)
LexToken(SINGLEEQUAL,'=',11,102)
LexToken(ID,'a',11,104)
LexToken(TIMES,'*',11,106)
LexToken(ID,'go',11,108)
LexToken(LPAREN,'(',11,110)
LexToken(ID,'b',11,111)
LexToken(COMMA,',',11,112)
```

```
LexToken(ID,'a',11,114)
LexToken(MINUS,'-',11,115)
LexToken(INTLITERAL,1,11,116)
LexToken(RPAREN,')',11,117)
LexToken(SEMICOLON,';',11,118)
LexToken(RCURLY,'}',12,124)
LexToken(ELSE,'else',12,125)
LexToken(LCURLY,'{',12,129)
LexToken(ID,'g',13,139)
LexToken(SINGLEEQUAL,'=',13,141)
LexToken(INTLITERAL,1,13,143)
LexToken(SEMICOLON,';',13,144)
LexToken(RCURLY,'}',14,150)
LexToken(TIMES,'*',15,156)
LexToken(ID,'b',15,157)
LexToken(SINGLEEQUAL,'=',15,159)
LexToken(TIMES,'*',15,161)
LexToken(ID,'b',15,162)
LexToken(PLUS,'+',15,164)
LexToken(ID,'g',15,166)
LexToken(SEMICOLON,';',15,167)
LexToken(ID,'k',16,173)
LexToken(SINGLEEQUAL,'=',16,175)
LexToken(ID,'k',16,177)
LexToken(PLUS,'+',16,179)
LexToken(ID,'g',16,181)
LexToken(SEMICOLON,';',16,182)
LexToken(RETURN,'return',17,188)
LexToken(ID,'g',17,195)
LexToken(SEMICOLON,';',17,196)
LexToken(RCURLY,'}',18,198)
LexToken(INT,'int',20,201)
LexToken(ID,'main',20,205)
LexToken(LPAREN,'(',20,209)
LexToken(VOID,'void',20,210)
LexToken(RPAREN,')',20,214)
LexToken(LCURLY,'{',21,216)
LexToken(ID,'k',22,222)
LexToken(SINGLEEQUAL,'=',22,224)
LexToken(INTLITERAL,0,22,226)
LexToken(SEMICOLON,';',22,227)
LexToken(ID,'f',23,233)
LexToken(SINGLEEQUAL,'=',23,235)
LexToken(ID,'go',23,237)
LexToken(LPAREN,'(',23,239)
LexToken(ADDR,'&',23,240)
LexToken(ID,'k',23,241)
LexToken(COMMA,',',23,242)
LexToken(INTLITERAL,5,23,244)
LexToken(RPAREN,')',23,245)
LexToken(SEMICOLON,';',23,246)
LexToken(ID,'printf',24,252)
LexToken(LPAREN,'(',24,258)
LexToken(STRINGLITERAL,'"%d\n"',24,259)
LexToken(COMMA,',',24,264)
LexToken(ID,'f',24,266)
LexToken(RPAREN,')',24,267)
LexToken(SEMICOLON,';',24,268)
```

```
LexToken(ID,'printf',25,274)
LexToken(LPAREN,'(',25,280)
LexToken(STRINGLITERAL,'"%d\n"',25,281)
LexToken(COMMA,',',25,286)
LexToken(ID,'k',25,288)
LexToken(RPAREN,')',25,289)
LexToken(SEMICOLON,';',25,290)
LexToken(RETURN,'return',26,296)
LexToken(INTLITERAL,0,26,303)
LexToken(SEMICOLON,';',26,304)
LexToken(RCURLY,'}',27,306)
```

3 语法分析

3.1 简介

语法分析是编译过程的一个逻辑阶段。语法分析的任务是在词法分析的基础上将单词序列组合成各类语法短语，如“程序”,“语句”,“表达式”等等. 语法分析程序判断源程序在结构上是否正确. 源程序的结构由上下文无关文法描述. 语法分析程序可以用 YACC 等工具自动生成。在本实验中我们使用了 ply.yacc 的语法分析工具，采用的算法是 LAIR(1)。程序执行的结果是一棵语法树，由 python 数据结构里的元组和列表来表示。

3.2 实现细节

我们实现的 minic 语言实现了 C 语言的大多数语法，不过出于简便，我们对 C 语言的语言规则进行了一定的限制。以下是我们的一些文法规则和解释：

```
program : declarationList
declarationList : declarationList declaration
                | declaration
declaration : staticVariableDeclaration
            | functionDeclaration
```

解释：在这里 program 是整一份代码的顶端入口，这是一个最顶层的文法规则。在这里声明列表是由一系列的声明所构成的，全局下的声明又是由函数和全局变量声明所构成的。

```
functionDeclaration : typeSpec ID LPAREN parameters RPAREN compoundStatement

typeSpec            : VOID
                    | INT
                    | FLOAT
                    | CHAR
                    | DOUBLE
                    | structSpecifier

structSpecifier     : STRUCT ID LCURLY staticVariableDeclarationList RCURLY
                    | STRUCT ID
```

解释：函数声明的文法如上所示，typeSepc 是函数的返回类型，ID 是函数的名字，parameters 是参数的列表，compoundStatement 是函数体。

```
compoundStatement : LCURLY optionalLocalDeclarations optionalStatementList RCURLY

whileStatement : WHILE LPAREN expression RPAREN statement

forStatement : FOR LPAREN optionalExpression SEMICOLON optionalExpression SEMICOLON
optionalExpression RPAREN statement

breakStatement : BREAK SEMICOLON
```

```
ifStatement : IF LPAREN expression RPAREN statement optionalElseStatement

returnStatement : RETURN expression SEMICOLON

continueStatement : CONTINUE SEMICOLON
```

解释：块语句的定义如上所示，在大括号里面包含了局部变量的声明以及局部逻辑控制语句。在这里列出我们实现的主要逻辑控制语句，分别是：if 条件判断语句、for 循环语句、while 循环语句、break 语句、continue 语句和 return 语句。

```
staticVariableDeclaration : typeSpec declaratorList SEMICOLON
                          | typeSpec declarator LSQUARE INTLITERAL RSQUARE SEMICOLON

declarator : pointer ID
           | ID

pointer : TIMES
        | TIMES pointer
```

解释：以上是静态变量声明的语句，这里的静态变量包含了不同变量的声明和数组的声明，出于简便起见我们的数组只允许单独声明，不能和普通变量出现在一个声明语句以内。我们同时也实现了指针的声明。

```
precedence = (
#('nonassoc','LESSTHAN', 'GREATERTHAN'),
('right','SINGLEEQUAL','PLUSEQUAL','MINUSEQUAL','DIVIDEEQUAL','TIMESEQUAL'),
('left', 'DOUBLEPIPES'),
('left', 'DOUBLEAMPERSAND'),
('left','BANGEQUAL','DOUBLEEQUAL'),
('left', 'LANGLE','RANGLE','LANGLEEQUAL','RANGLEEQUAL'),
('left', 'PLUS', 'MINUS'),
('left', 'TIMES', 'DIVIDE', 'PERCENT'),
('left', 'LSQUARE'),
('right', 'UMINUS','EXCLAMATION','REMOVEREF','ADDR'),
('left','ELSE')
)
```

解释：由于 expression 的定义较为复杂，我们首先确定了它的优先级以及结合的顺序

```
expression : expression PLUS expression
           | expression MINUS expression
           | expression TIMES expression
           | expression DIVIDE expression
           | expression DOUBLEPIPES expression
           | expression DOUBLEAMPERSAND expression
           | expression LANGLE expression
           | expression RANGLE expression
           | expression LANGLEEQUAL expression
           | expression RANGLEEQUAL expression
           | expression BANGEQUAL expression
           | expression DOUBLEEQUAL expression
           | expression PERCENT expression
```

解释：上面是二元运算表达式的文法，我们在这里实现了加、减、乘、除等基本的二元运算。

```
expression : expression DOT ID
expression : expression LEFTARROW ID
expression : ADDR expression
expression : REMOVEREF expression
```

解释：以上是和指针以及结构体相关的文法，在这里我们实现了取地址、解引用的运算，并且实现了结构体变量以及结构体指针访问成员的运算。

```
expression : expression DIVIDEEQUAL expression
           | expression PLUSEQUAL expression
           | expression MINUSEQUAL expression
           | expression TIMESEQUAL expression

expression : expression DOUBLEPLUS
           | expression DOUBLEMINUS
```

解释：以上是基于 C 语言特性的运算表达式，这里实现了加等于、减等于之类的压缩表示运算以及自加和自减的运算。

3.3 测试结果

考虑如下 C 语言代码：

```c
int ans;

int gcd(int a, int b)
{
    int g;
   if(b==0){
        g = a;
    }else{
        g = gcd(b, a % b);
    }
    return g;
}

int main(void)
{
    ans = gcd(9, 36) * gcd(3, 6);
    printf("%d\n", ans);
    return 0;
}
```

它的语法结构在 python 的表达如下：

```
('program', ('declarationList', [('declaration', ('staticVariableDeclaration',
('typeSpec', 'int'), ('declaratorList', [('declarator', 'ans')]))),
('declaration', ('functionDeclaration', ('typeSpec', 'int'), 'gcd',
('parameters', ('parameterList', [('parameter', ('typeSpec', 'int'),
('declarator', 'a')), ('parameter', ('typeSpec', 'int'), ('declarator', 'b'))])),
('compoundStatement', ('optionalLocalDeclarations', ('localDeclarations',
[('localDeclaration', ('staticVariableDeclaration', ('typeSpec', 'int'),
('declaratorList', [('declarator', 'g')]))])), ('optionalStatementList',
('statementList', [('statement', ('ifStatement', ('binary-expression', '==',
('identifier-expression', 'b'), ('int-expression', 0)), ('statement',
('compoundStatement', ('optionalLocalDeclarations', None),
('optionalStatementList', ('statementList', [('statement', ('expressionStatement',
('assignExpression', ('identifier-expression', 'g'), ('identifier-expression',
'a'))))])))), ('optionalElseStatement', ('statement', ('compoundStatement',
('optionalLocalDeclarations', None), ('optionalStatementList', ('statementList',
[('statement', ('expressionStatement', ('assignExpression',
('identifier-expression', 'g'), ('functioncallExpression', 'gcd',
('argumentExpressionList', [('identifier-expression', 'b'), ('binary-expression',
'%', ('identifier-expression', 'a'), ('identifier-expression', 'b'))])))))])))))))),
```

```
('statement', ('returnStatement', ('identifier-expression', 'g')))])))))),
('declaration', ('functionDeclaration', ('typeSpec', 'int'), 'main', ('parameters',
'void'), ('compoundStatement', ('optionalLocalDeclarations', None),
('optionalStatementList', ('statementList', [('statement', ('expressionStatement',
('assignExpression', ('identifier-expression', 'ans'), ('binary-expression', '*',
('functioncallExpression', 'gcd', ('argumentExpressionList', [('int-expression', 9),
('int-expression', 36)])), ('functioncallExpression', 'gcd', ('argumentExpressionList',
[('int-expression', 3), ('int-expression', 6)])))))), ('statement', ('expressionStatement',
('functioncallExpression', 'printf', ('argumentExpressionList',
[('stringExpression', '"%d\n"'), ('identifier-expression', 'ans')])))),
('statement', ('returnStatement', ('int-expression', 0)))])))))]))
```

4 语义分析

4.1 简介

语义分析是编译过程的一个逻辑阶段，语义分析的任务是对结构上正确的源程序进行上下文有关性质的审查，进行类型审查。语义分析是审查源程序有无语义错误，为代码生成阶段收集类型信息。比如语义分析的一个工作是进行类型审查，审查每个算符是否具有语言规范允许的运算对象，当不符合语言规范时，编译程序应报告错误。如有的编译程序要对实数用作数组下标的情况报告错误。又比如某些程序规定运算对象可被强制，那么当二目运算施于一整型和一实型对象时，编译程序应将整型转换为实型而不能认为是源程序的错误。

我们没有特别进行繁多的类型审查处理，而是只支持整数 int 类型的算数操作。我们实现的语义分析主要是为中间代码生成做准备工作，包括语法树的语义层面的解析和符号表的维护。

我们采用了语义分析和中间代码生成同时进行的方法，这里只介绍语义分析的基本方法和实现细节。

4.2 基本方法

将语法树按语句类型逐级分层，利用函数调用栈递归解析。

(1) 例如下面分析语句的函数，根据语法树不同语句的语法规则，按类型分为：

if 语句、表达式语句、函数返回语句、for 语句、while 语句、复合语句、break 语句、continue 语句等

并分别调用不同的解析函数完成递归解析。

```
def parse_statement(statement,label_start="continue???",label_end="break???"):
        assert(statement[0]=="statement")
        assert(len(statement)==2)
        if statement[1][0]=="ifStatement":
             parse_ifstatement(statement[1],label_start,label_end)
        elif statement[1][0]=="expressionStatement":
             ret_temp_var(parse_expressionStatement(statement[1]))
        elif statement[1][0]=="returnStatement":
             parse_returnStatement(statement[1])
        elif statement[1][0]=="forStatement":
             parse_forStatement(statement[1])
        elif statement[1][0]=="whileStatement":
             parse_whileStatement(statement[1])
        elif statement[1][0]=="compoundStatement":
             parse_compoundStatement(statement[1],label_start,label_end)
        elif statement[1]=="breakStatement":
             print("\t"+str(("j",None,None,label_end)))
        elif statement[1]=="continueStatement":
             print("\t"+str(("j",None,None,label_begin)))
        else:
             print(statement[1])
             assert(False)
        return
```

(2) 又有如下关于解析声明的函数，按声明的类型分为：
函数声明、静态变量声明、静态变量数组声明等
然后递归调用不同的解析处理函数完成解析。

```
def parse_declaration(declaration):
        assert(declaration[0]=="declaration")
        if declaration[1][0]=="functionDeclaration":
              parse_functionDeclaration(declaration[1])
        elif declaration[1][0]=="staticVariableDeclaration":
              parse_staticVariableDeclaration(declaration[1])
        elif declaration[1][0]=="staticVariableArrayDeclaration":
              parse_staticVariableArrayDeclaration(declaration[1])
        else:
              assert(False)
```

4.3 实现细节

具体实现上，在递归调用解析函数的过程中对语句的主要信息解析记录和分析，以静态变量的声明为例：

当遇到一条如下 c 代码时，语法分析器会生成包含”int a;” 声明的声明表 declaration list。

```
int a;
```

语义分析器 (即与中间代码生成器整合在一起的工具) 按照如下处理栈对其进行分析。

```
parse_declarationList ->
parse_declaration ->
parse_staticVariableDeclaration
```

最后在 parse_staticVariableDeclaration 函数中将该静态变量记录在全局变量符号表 global_var 中。

```
def parse_staticVariableDeclaration(staticVariableDeclaration):
        # ...
        global global_var
        var_type=staticVariableDeclaration[1][1]
        for _ in staticVariableDeclaration[2][1]:
              if len(_)==2:
                    global_var.append((var_type,_[1],(False,1),None))  # int a
              else:  # pointer
                     global_var.append((var_type+_[1][1][0],_[2],(False,1),None))
                       # int *a
        return
```

4.4 测试结果

5 符号表设计

5.1 简介

主要维护了两类符号，全局变量 global_var，和函数中的局部变量。函数中的局部变量又分为函数参数 local_para 和函数内定义的变量 local_var。全局定义的函数又作为一种全局的函数变量。

由于我们采用的是，符号表与中间代码一起生成的方法。在此文档中仅介绍符号表生成的部分。

5.2 实现细节

(1) 采用 Python 字典数据结构 dict 存储符号表 symbol_table。

```
symbol_table = {}
```

(2) 在分析 declarationList 之后得到全局变量，将其存储到符号表中

```
def parse_declarationList(declarationList):
        assert(declarationList[0]=="declarationList")
        for _ in declarationList[1]:
              parse_declaration(_)
        global global_var
        print("global_var =",global_var)
        global symbol_table
        symbol_table.setdefault('global', {}).setdefault('global_var', global_var)
```

(3) 在分析函数参数时，将参数符号存储到符号表

```
def parse_parameters(parameters, label_func):
        # ...
        parse_parameterList(parameters[1])
        # ...
        global symbol_table
        symbol_table.setdefault(label_func, {}).setdefault('local_para', local_para)
```

(4) 在分析函数定义之后得到函数内定义的局部变量，将其存储到符号表

```
def parse_functionDeclaration(functionDeclaration):#1-> return type 2->name 3->para 4->body
# ...
        parse_parameters(functionDeclaration[3], label_func)
        assert(functionDeclaration[4][0]=="compoundStatement")
# ...
        global symbol_table
        symbol_table.setdefault(label_func, {}).setdefault('local_var', local_var)
```

6 代码生成

6.1 中间代码生成

生成的中间代码是自定义的四元式, 形如 ‘(op,src1,src2,dest)‘

6.1.1 四元式

四元式内容说明

- **op** 说明指令的大致运算类型, 它可以是某个双目运算符, 或是某种赋值运算符, 也可以是跳转指令, 还可以是一些函数相关的汇编指令, 如”call”(表示调用),”arg”(表示函数参数压栈),”ret”(表示函数返回);
- **src** 说明操作数, 它可以某个变量, 或是某个常数, 还可以是常数名字, 特定情况下可以为空;
- **dest** 说明的是操作的目标, 通常是一个变量或是变量组 (用于说明给数组元素赋值), 也可以是标签 (用于跳转), 特定情况下可以为空.

四元式中的作为操作数的变量不仅仅只有变量名这一个信息, 而是作为一个元组, 还带有额外信息用于辅助生成汇编代码.

操作数形如 ‘(var_name,(is_global_or_temp,type,byte_addr))‘

其中:

- **var_name** 是一个简单 python 字符串用以说明的变量名字, 实际上在生成汇编的过程中只有使用全局变量时才会用到这个信息;
- **is_global_or_temp** 标记变量是否是全局变量或是局部的临时变量, 取值为 **True** 或 **False**, 或是一串 python 的 bytes(说明是字符串);
- **type** 是一个简单 python 字符串用以说明的变量类型, 如”**int**”, ”**int***”, ”**double**”, ”**char**”, ”**float**” 等;

- **byte_addr** 指的是变量的偏移, 不过参考的基准略有不同:
  - 普通全局变量无意义, 设为 None
  - 字符串常量,
  - 函数局部参数, 设置为相对于" **返回地址**"(**return address**) 的偏移 (所以是非负数)
  - 局部变量 (非临时变量), 相对于帧指针 **ebp** 的偏移地址 (所以是负数)
  - 局部变量 (临时变量), 相对于函数在栈中挖出当前函数所有变量空间后的栈指针 **esp** 的偏移地址 (所以是正数)

下方对若干自定义四元式举例说明:

```
;i=0;给局部变量 i 赋值 0,i 的基地址在 [ebp-44],数据宽度是 sizeof(int)=4
(':=', 0, None, ('i', (False, 'int', -44)))

;temp_var_4=(i<10);其中 temp_var_4 是临时变量,假设函数所有变量的大小为 x(Byte),那么 temp_var_4 基地
('<', ('i', (False, 'int', -44)), 10, ('temp_var_4', (True, 'int', 0)))

;if(temp_var_4==0)goto main_L2;其中 main_L2 是一个标签
('j==', ('temp_var_4', (True, 'int', 0)), 0, 'main_L2')

;参数 aFDDN 的地址压栈
('arg', ('aFDDN', (b'"f[%d]=%d\\n"', 'char[]', 13)), None, None)

;调用函数 printf;已经实现将参数压栈,printf 的返回值实际上是成功输出的字符个数,返回值最终存入 temp_va
('call', 'printf', None, ('temp_var_4', (True, 'int', 0)))

;temp_var_4=&k;k 是全局变量数据类型是 int,temp_var_4 是临时变量,数据类型int
(':=(&)', ('k', (True, 'int', None)), None, ('temp_var_4', (True, 'int', 0)))

;return g;其中 g 是局部变量,实际地址为 [ebp-4],数据宽度 sizeof(int)=4
('ret', ('g', (False, 'int', -4)), None, None)
```

6.1.2 函数调用

这里特殊列出来是因为两个原因:

- 函数调用的时候有个针对字符串常量的特殊处理
- 函数调用协议有很多种:**__stdcall**,**__cdecl**,"**__pascall**","**__thiscall**" 以及 **__fastcall**, 实现函数调用在默认情况下必须统一为其中某一种.

函数调用中的字符串常量 考虑这样一个代码

```
printf("helloworld\n");
```

在函数调用的过程中, 正常的编译器不会把整个字符串压入堆栈 (, 除非 setbuf 清空了特定缓冲区), 而是仅仅压入字符串的地址. 但是为什么某个地址会恰好有这个字符串呢? 答案就是, 编译器在数据段 (全局变量区域) 有意构造了这样的字符串.

但是给这个字符串取名字也有点问题, 毕竟最终的代码我们并不在意它的名字, 但是在写汇编的过程中我们需要用符号助记它的基地址, 这个名字又不能和其他全局变量重名... 最终是参考了 **IDA Pro** 逆向可执行文件的时候给字符串命名的方式, 自己设计了一个命名机制.

这个串会导致在数据段额外增加一些数据:

```
data segment
        aHELLOWORLDN "helloworld",0ah,0dh,0,'$'
data ends
```

由于汇编没有字符串转移, 表示”
n” 这个回车字符实际上将其转化为 ascii 码 0ah 就行了. 至于后面还有 0dh, 这是因为 Windows 中,”
n” 实际上是执行了**回车 (光标移到所在行行首)** 和**换行 (光标下移一位)** 两个操作. 再后面的 0 是 c/c++ 字符串的结束标志. 再在之后的’$’ 是 dos 模式下字符的结束标志, 放在这里是为了一定程度上避免被解释为 dos 下的 exe 运行的时候会出大问题.

函数真正调用的时候会被编译成:

```
push offset [aHELLOWORLDN]
call crt_printf;msvcrt.lib 库的 printf 就叫这个名字
```

函数调用协议 调用协议的区别在网上有很多阐释, 此处着重说明我们使用的 **__stdcall**

- 参数只能通过堆栈传递, 进入函数时不会有寄存器储存参数;
- 函数的压栈顺序是从右到左的;
- 函数结束后, 被调用者负责清理栈空间 (函数参数占据的栈空间).

前面两个比较好理解, 关于最后一个, 一两份代码进行说明: **被调用者负责清理栈空间示例**

```
code segment
f proc
        push ebp
        mov ebp,esp
        mov eax,[ebp+8]
        inc eax
        mov esp,ebp
        pop ebp
        ret 4
f endp
main proc
        push ebp
        mov ebp,esp
        xor eax,eax
        push eax
        call f
        mov esp,ebp
        pop ebp
main endp
code ends
end main
```

可以看到, 被调用者清理栈空间,f 函数最后是用 ‘ret 4‘返回的, 说明找到返回地址之后, 堆栈还得弹出 4B 的空间再返回. 与之相对的, 调用者清理栈空间,f 函数最后是用 ‘ret‘返回的, 找到返回地址直接跳转. 在 main 函数的 call 之后, 还好有一个恢复堆栈的 add 指令.

由于 **__stdcall** 适用于 Win API, 我们就延续此传统使用了 **__stdcall**.

6.1.3 变量定义 (符号表)

由于全局变量能在函数体以外的任何地方定义, 我们需要扫描整个语法树在能知道所有的全局变量, 因此在所有函数段结束后, 输出一个 list 说明全局变量的信息

在函数的中间代码开始之前, 输出 list 说明函数的参数信息. 又由于函数体内任何地方都可以定义变量, 在函数结束后再输出所有局部变量的信息.

至于这些变量的信息, 都是 list 中的一个个元组列举出来的, 形如 ‘(type,name,(is_array,size),offset)‘

元组的元素说明如下:

- **”type”** 这是一个 python 字符串说明标识符类型
- **”name”** 是一个 python 字符串说明标识符名字

- **"is_array"** 是一个二值变量说明标识符是否对应数组
- **"size"** 说明标识符占用空间大小 (以标识符类型为单位), 即 'int a[10]',a 在此处的值为 10
- **offset** 偏移地址, 全局变量此处为 None, 局部变量是相对于 ebp 的位置, 函数参数是相对于 ebp+8(8 表示有 4B 的 ebp 和 4B 的返回地址) 的偏移

局部变量地址分配 考虑这样一段函数:

```
int main(){
    int i;
    int f[10];
    for(i=0;i<10;i++)
        if(i%2==0){
            int j;
            j=i*2;
            f[i]=j/2;
        }
        else{
           int k;
           k=i*3;
           f[i]=k/2;
        }
        return 0;
}
```

函数在不同的大括号中定义了不同的新的局部变量, 逆向编译出来的程序发现, 编译器是给 j 和 k 都开辟了特定的栈空间, 也就是说 j 和 k 对应的地址是没有交叉的.

这也大大方便了我们程序的编写, 最终我们做到了编译出来的地址分配的效果与下方的程序是完全一致的.

```
int main(){
    int i;
    int f[10];
    int j;
    int k;
    reutrn 0;
}
```

那么假如将之前函数中的 k 换成 j 会怎么样呢? 我们是碰到定义才把相应的标识符加到局部变量 list 的队首, 每次使用的时候又从前往后匹配, 这样一来保证了作用域的正确性 (之前 j 的作用域在大括号结束后会被再之后的 k 的作用域所覆盖), 因此换了之后可以保证地址不出问题.

6.2 目标代码生成

最终期望生成码 masm32 汇编, 能够在 Windows 操作系统上编译连接并运行, 最终也是实现了这个要求.

6.2.1 代码头

由于需要利用其它库实现输入输出, 因此需要用汇编导入库 (保证 **msvcrt.inc** 和 **msvcrt.lib** 和代码在同一个目录)

```
.386
.model flat,stdcall
include msvcrt.inc
includelib msvcrt.lib
```

6.2.2 数据段

显然我们只用处理全局变量, 我们在中间代码处理完程序后最后输出了一个描述全局变量的 list, 分析后转成汇编代码即可, 如:

```
data segment use32
        data_f dd 0
        data_k dd 0
        aD db "%d",0ah,0dh,0,'$'
data ends
```

6.2.3 代码段

将每条中间代码转化成若干条汇编指令以实现功能即可. 一般来说操作都是从内存中取出数据到寄存器, 寄存器进行运算后将数据写回另一个地址.

有个值得注意的小地方是,mian 函数处理有些不同,masm32 下 ‘return 0‘实际上会回到更高层的调用我们可执行文件的程序, 程序不会结束, 最终暴力的用 ‘exit(0)‘结束整个程序. 除此之外, 代码段也需要指定程序入口, 代码最后一行 ‘end main‘就是用伪指令完成了这个操作.

7 优化考虑

7.1 中间代码优化

7.1.1 常数折叠

在由语法树转化为中间代码的过程中, 进行常数折叠, 亦即在编译期间处理出一定能够计算出来的常数表达式的值, 如:

```
int a;
a=1+2+3;
```

生成中间代码的时候未经优化会被解释为类似下方四元式的中间代码 (注意,** 四元式的第一个参数表示运算符, 第二三个表示操作数, 即右值, 最后一个表示目标位置, 即左值 **):

```
("+",1,2,temp_var_8)
("+",temp_var_8,3,temp_var_4)
(":=",temp_var_4,None,a)
```

”temp_var_” 开头的表示编译器自动生成的中间变量, 它们会占用栈空间. 但是显然这样解释的话会给运行的程序增加压力, 因为浪费了很多不必要的栈空间和 CPU 的运算资源. 因此在优化后的生成代码的程序中, 在处理到某个语法树的节点的时候, 一旦发现整颗子树表示普通的常数表达式, 我们构造出原来的常数表达式, 使用 Python 自带的 **eval()** 函数求值即可, 代码可以优化为:

```
(":=",6,None,temp_var_4)
(":=",temp_var_4,None,a)
```

7.1.2 堆栈使用优化

在堆栈空间无限的情况下, 为每个局部表达式生成一个临时变量的空间是没有问题的. 但是在 c/c++ 中, 堆栈资源有限 (不过在测试程序中体现不出来), 大量使用堆栈会使得在递归的时候更容易出现**爆栈**. 以下方函数中的代码为例:

```
int a,b,c,d,e;
a=b+c+d+e;
```

未经优化的话, 程序会为 (b+c), (b+c+d), (b+c+d+e) 这三个表达式都额外开辟一份内存空间, 如下方中间代码的示例:

```
("+",b,c,temp_var_12)
("+",temp_var_12,d,temp_var_8)
("+",temp_var_8,e,temp_var_4)
(":=",temp_var_4,None,a)
```

观察可以看到, 将代码分割为前两行和后两行, 前两行中并未使用过 ***temp_var_4***, 后两行中从未使用过 ***temp_var_12***. 两者的使用范围没有交集, 我们可以让这两个临时变量共享栈里面的同一个位置, 优化后的中间代码如下所示:

```
("+",b,c,temp_var_8)
("+",temp_var_8,d,temp_var_4)
("+",temp_var_4,e,temp_var_8)
(":=",temp_var_8,None,a)
```

具体的实现方法也很直观, 我们设计了一个可以支持无限中间变量的**变量池**, 每次需要使用中间变量的时候, 从**变量池**中取出下标最小的中间变量, 使用完后如果保证之后不再使用就归还回**变量池**.

至于判定是否是" 之后不再使用" 的规则, 对于任意一个四元式, 我们记为 '(op,oprand1,oprand2,dest)', 一般来说我们做的运算是 'dest := oprand1 op oprand2' 之后 dest 有可能再参与之后的运算, 因此 'oprad1'和 'oprand2'的生命周期到此结束, 如果他们是临时变量的话就可以归还回**常量池**了. 唯一一个例外的地方是赋值操作 '(":=",oprand1,None,dest)', 进行赋值操作后, 可能继续使用 'oprand1'于是试图归还 (不是临时变量就不能归还)'dest'.

7.2 汇编代码优化

编译器并未实现中间代码转成汇编代码的过程中进行优化, 这里**汇编代码优化**是指生成汇编代码后直接针对汇编代码进行优化.

7.2.1 " 脏" 寄存器赋值

此处先解释" 脏"(dirty) 的概念. 在汇编语言中主要的数据是存储在内存中的, 但是几乎没有汇编指令支持多个操作数全是内存. 这两者之间的冲突需要 CPU 里面的寄存器进行过渡. 因此从某种程度上来说, 寄存器是内存的拷贝. 但是这个时候又出现了一个情况, **寄存器内容发生改变的时候, 不会立即同步到内存中, 我们把这种和对应内存不一致的寄存器成为" 脏的"**.

显然对于不" 脏" 的寄存器, 我们没有必要用对应的内存进行多次赋值, 如:

```
mov dword ptr [ebp-4+0],eax
mov eax,dword ptr [ebp-4+0]
```

第二条指令是无论如何都不需要的, 因此省略为:

```
mov dword ptr [ebp-4],eax
```

7.2.2 寄存器作为源 (src)

此过程不会减少代码行数, 但是有两个作用:

- 对汇编代码进行常数优化, 因为寄存器总是比内存访问快
- 能够提高之后写入内存的优化程度

考虑这样若干条指令:

```
mov dword ptr [ebp-4+0],edi
mov eax,dword ptr [ebp-4+0]

mov dword ptr [ebp-4+0],edi
mov eax,edi

mov dword ptr [ebp-4+0],eax
push dword ptr [ebp-4+0]

mov dword ptr [ebp-4+0],eax
push eax
```

7.2.3 省略部分写入内存

进行了之前的优化后, 仍然存在大量寄存器向内存的不必要赋值. 不必要需要满足以下两个条件:

- 是寄存器向临时变量赋值
- 在 block 内, 之后的指令中, 不会再将临时变量的值作为源, 或在作为源之前会再次被赋值

P.S. block 定义为不含跳转指令也没有标签的连续的尽可能长的代码段, 唯一的例外是 block 的最后一条指令可以是跳转指令. 之所以只考虑一个 block 的情况是因为生成未优化的汇编代码时保证一个 block 里面一定是先对内存赋值, 之后才可能将其作为源. 考虑这样一个代码段:

```
ll0:
      mov dword ptr [ebp-4+0],edi
      mov eax,edi
      cmp eax,0
      jnz ll1
      jmp ll2
ll1:
     ;...
ll2:
     ;...
```

标签 **ll0** 到 **jnz** 指令 (含) 的代码构成一个 block, 但是我们可以明显看到第一条指令给 'dword ptr [ebp-4+0]'赋值后就再也没有用到, 而且地址的格式是 '[ebp-x+y]'(我们保证这种形式的地址是中间变量), 说明这是个编译器生成的中间变量, 我们依据之前的规则可以省略第一条赋值指令.

8 测试案例

由于测试内容过多，在报告里省略不表，具体见文件夹 Doc/samples

9 其他设计

9.1 语法树可视化

语法树是句子结构的图形表示，它代表了句子的推导结果，有利于理解句子语法结构的层次。简单说，语法树就是按照某一规则进行推导时所形成的树。

通常语法树是一颗多叉树，需要采用多叉树的数据结构进行存储。

9.1.1 树形数据结构的构建

(1) 在 Python 中定义 Node 类，包含两个变量 children 存储所有子节点，value 存储当前节点的值

```
class Node:
       def __init__(self, c=[], v=None):
              self.children = c
              self.value = v
```

(2) 从 yacc 生成的元组表示的语法树构造树形结构

```
def buildtree(root):
       if type(root)!=tuple:
            return Node([], root)
       node = Node([], root[0])
       if type(root[1]) == list:
            for x in root[1]:
                   node.children.append(buildtree(x))
       else:
              for i in range(1, len(root)):
                    node.children.append(buildtree(root[i]))
       return node
```

(3) 将语法树树形结构转化为 python 的 dict 表示，以符合可视化库 graphviz 的输入格式

```
def tree2dict(root):
       if len(root.children)==0:
            return str(root.value)
       res = {root.value: {}}
       cnt = 0
       for c in root.children:
            res[root.value].setdefault(cnt, tree2dict(c))
            cnt+=1
       return res
```

9.1.2 采用 graphviz 中的 Digraph 可视化树

由于多叉树是一种特殊的图，可以采用 graphviz 图可视化工具进行可视化。
(1) 首先创建 Digraph 对象。

```
g = Digraph("G", filename=name, format='png', strict=False)
```

(2) 利用上述 dict 定义其边和结点

```
def _sub_plot(g, tree, inc):
       global root

       first_label = list(tree.keys())[0]
       ts = tree[first_label]
       for i in ts.keys():
            if isinstance(tree[first_label][i], dict):
                 root = str(int(root) + 1)
                 g.node(root, list(tree[first_label][i].keys())[0])
                 g.edge(inc, root, str(i))
                 _sub_plot(g, tree[first_label][i], root)
            else:
                root = str(int(root) + 1)
                g.node(root, tree[first_label][i])
                g.edge(inc, root, str(i))
```

(3) 调用可视化接口

```
g.view()
```

9.1.3 例子和结果

```
d1 = {"no surfacing": {0: "no", 1: {"flippers": {0: "no", 1: "yes"}}}}
```

9.2 错误处理

9.2.1 语法错误的处理

主要通过 yacc 的 error 处理函数实现，当检测到不在定义中的语法规则时，yacc 自动调用 p_error 函数，并以当前处理的 token 作为参数。

```
def p_error(p):
       print('Syntax Error occur around', 'line:', p.lineno)
       print('Syntax Error occur around', p.type,':', p.value)
       exit()
```

上述语法错误处理函数将 token 中记录的当前行号 lineno 和处理到的符号类型 type 和值 value 进行格式化打印，并调用 exit() 结束语法分析。

9.2.2 语义错误的处理

由于我们采用了中间代码生成和语义分析一起处理的方法，语义错误的处理在生成中间代码的过程中处理错误。

具体做法是在分析的过程中通过 Python 的 assert 函数指明当前必须满足的条件，并在条件不满足时执行相应的错误处理函数。

例如，下面的 find_addr 函数是语义分析和中间代码生成过程中寻找变量地址的函数。当表达式既不在当前局部变量表和局部参数表中，也不在全局变量表中时，打印一个语义错误信息，并抛出错误。这意味着在当前环境下使用的该变量没有声明。

```
def find_addr(expression):
        if type(expression)!=str:
             return (None,None,None)
        global local_var,local_para,global_var
        for i in range(len(local_var)):
              if local_var[i][1]==expression:
                   return (False,local_var[i][0],local_var[i][3])
        for i in range(len(local_para)):
              if local_para[i][1]==expression:
                   return (False,local_para[i][0],local_para[i][3])
        for _ in global_var:
              if _[1]==expression:
                   return (True,_[0],None)
        print('Semantic Error: Cannot find address for expression:', expression)
        assert(False)
```

9.2.3 语法错误测试结果

```
int i;

int main(void)
{
    int a, b
    a = 0;
    b = 2;
    for(i=0;i<10;i=i+1){
    a = a + b;
}
    printf("%d\n", a);
    return 0;
}

Syntax Error occur around line: 6
Syntax Error occur around ID : a
```

9.2.4 语义错误测试

```
int i;

int main(void)
{
    int a, b;
    a = 0;
    b = 2;
    for(i=0;i<10;i=i+1){
        a = a + c;
    }
    printf("%d\n", a);
    return 0;
}
```

```
main proc
local_para = []
(':=', 0, None, ('a', (False, 'int', -8)))
(':=', 2, None, ('b', (False, 'int', -4)))
(':=', 0, None, ('i', (True, 'int', None)))
main_L1:
('<', ('i', (True, 'int', None)), 10, ('temp_var_4', (True, 'int', 0)))
('j==', ('temp_var_4', (True, 'int', 0)), 0, 'main_L2')
Semantic Error: Cannot find address for expression: c
```

上述结果表明在中间代码生成的过程中发现了错误。