实验二　语法分析器的实现及中间代码的生成

软工 1403　　2014010919　蔡嘉豪

一、　实验目的

使用 C++编写一语法分析器，使其可以对 C 语言的子集(ANSI C 标准，支持共 35 个关键字，详见实验一中的附录一)进行语法分析及生成中间代码。该语法分析器接收词法分析器分析得到的词法单元流，输出抽象语法树与中间代码及错误信息。

二、　语法分析器的设计

1) 整体设计

该语法分析器被视为是编译器整体的一部分，接收词法分析得到的词法单元流，通过语法分析将词法单元流转换为抽象语法树，最后，由抽象语法树内部结点调用虚函数 code_gen()来自动生成中间代码。

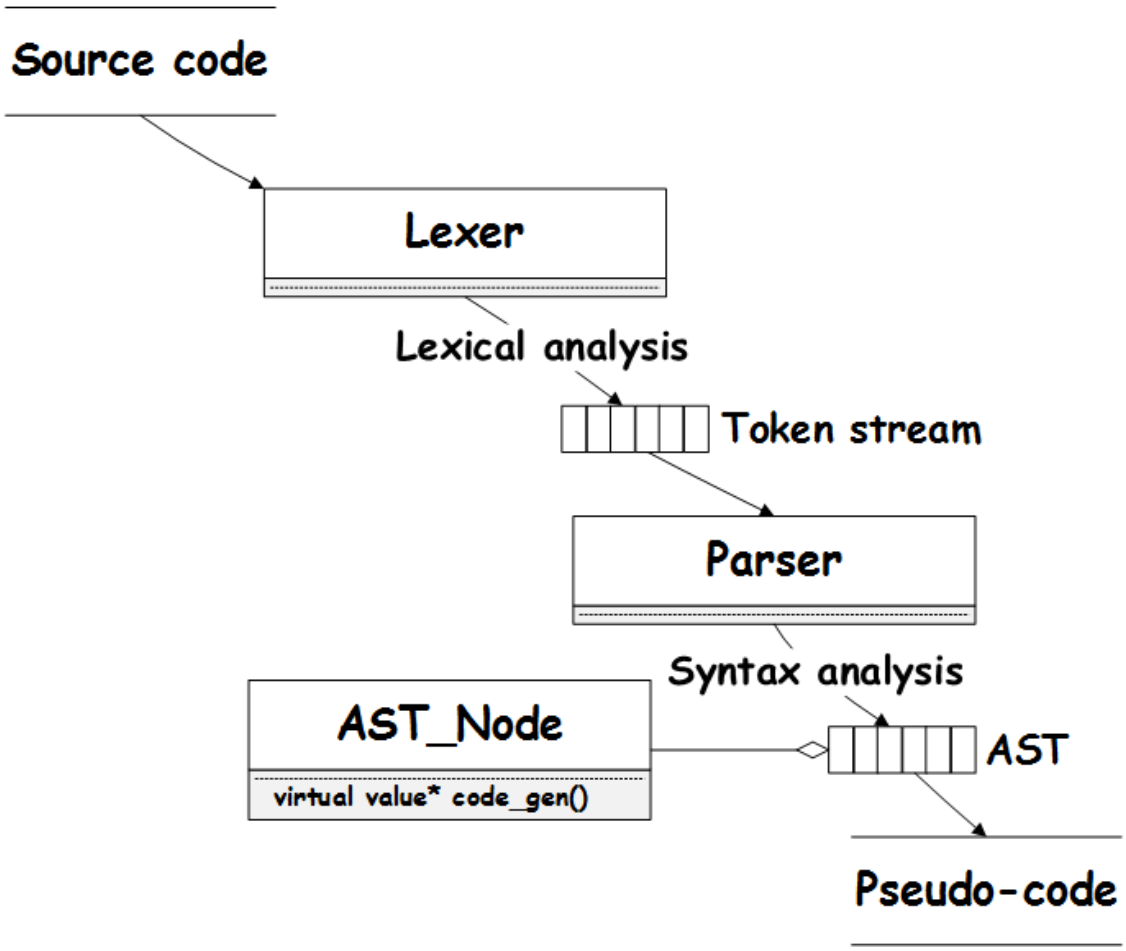

图 1 编译器整体设计

语法分析器循环读入词法单元流，不同的词法单元将对应不同的处理方案。例如：

- 读到 for 关键字时，应当跳转到处理 for 循环的函数中；
- 读到分号时，应当直接跳过该词法单元并处理下一个词法单元。

2) 抽象语法树的设计

抽象语法树由各种各样的语法树结点组成，语言中的每种结构都应该与某一种特定的语法树结点相对应，所有节点均继承自 expr_node。该语法分析器可以分析 if 语句、for 语句、二元表达式以及常量与变量，故共有 5 个结点继承自 expr_node。具体如图 2 所示。

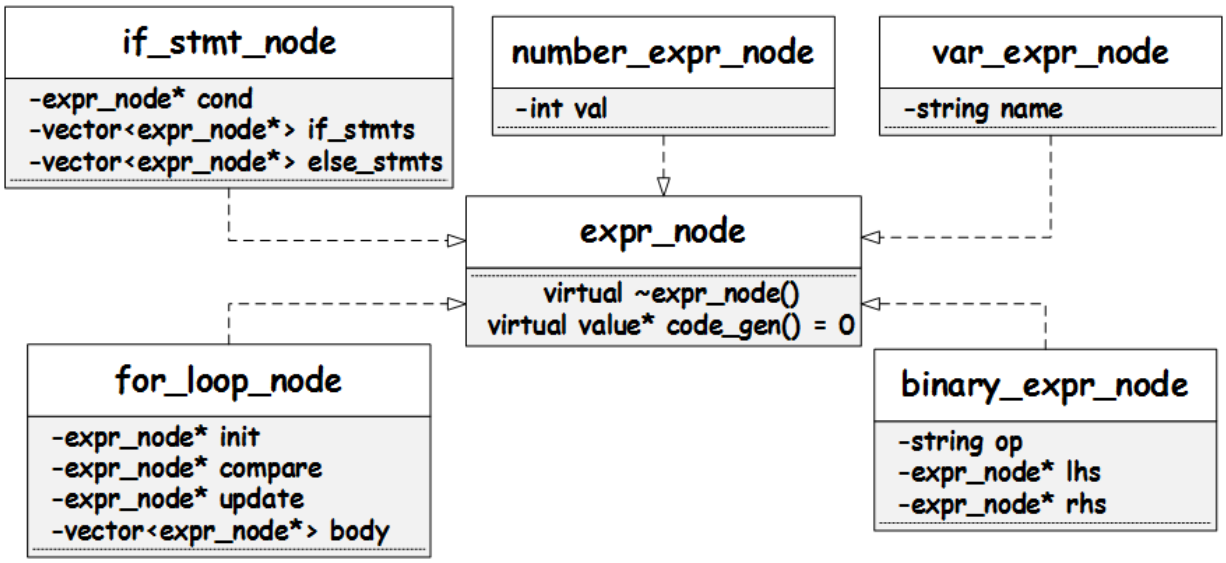

图 2 抽象语法树结点设计

上述结点的具体含义如图 3 所示：

结点名称	结点属性	属性含义
if_stmt_node	cond	if语句中的条件语句
	if_stmts	if语句块中的语句集合
	else_stmts	else语句块中的语句集合
	init	for循环中的初始化语句
for_loop_node	compare	for循环中的条件语句
	update	for循环中的更新语句
	body	for循环体中语句集合
number_expr_node	val	常量的值
var_expr_node	name	变量的名称
binary_expr_node	op	二元运算符
	lhs	运算符左部
	rhs	运算符右部

图 3 抽象语法树结点属性含义

3) 中间代码生成的解决方案

中间代码的生成主要依靠每个语法树结点中的 code_gen()函数。前面提到过每一种语法树结点都继承自 expr_node，expr_node 的定义如图 4 所示，只由两个虚函数组成。

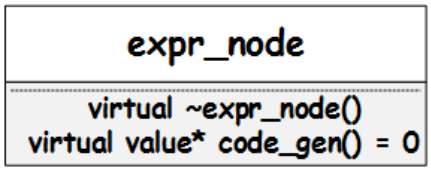

图 4 语法树结点基类

其中值得注意的是纯虚函数 code_gen()，所有语法树结点都继承自 expr_node，也就是说所有的语法树结点都必须重新实现各自的 code_gen()函数，该函数的作用是生成各自的中间代码。这样一来，每当语法分析器得到一个结点，就可以调用该结点对应的 code_gen()函数，从而得到该结点的中间代码。

三、 语法分析器的实现

1) 准备工作

- 平台： Ubuntu 15.04
- 语言： C++ (C++11 标准)
- 编译工具： GNU GCC 5.1.1

2) 语法分析的实现

因语法分析的种类过多而篇幅有限，且处理模式基本类似，故只选取其中一部分讲解。

1. 错误处理

首先，语法分析器必须具有错误处理机制，该语法分析器的错误处理策略如下：发生错误时不停止解析，报告错误原因与错误位置，不停跳过当前词法单元，直到当前词法单元是可以被已有函数所处理的词法单元。

如图 5 所示，该代码中有错误，在 for 循环中遗漏了右括号，语法分析器找到了这个错误并报告，然后跳到主控程序，开始寻找能够解析的词法单元。很明显，该语法分析器并不能解析以左大括号开头的语句，于是丢弃该词法单元，并找到下一个词法单元 a，此时将解析以 a 开头的表达式。但是由于跳过了左大括号而直接读到了又大括号，语法分析器再次报错。最后，它正确地解析了最后一句 first=5;。

```
Time:01:22:37     User:james     Dir:/mnt/hgfs/For-Linux/LambCompiler/bin
$./compiler
Lexer Test: 97/97 (100.00%) passed
    source code:
        x = a + b * c / (d * 5) - k;
        if (c == 10) {
            a = a + b;
        } else {
            b = b + a * (d - c);
        }
        for(i = 0; i == 5; i = i + 1 {
            a = a + i;
        }
        first = 5;

    pseudo-code:
test_dummy:7:37 : Error: expected ')' in for-loop
        for(i = 0; i == 5; i = i + 1 {
                                    ^
test_dummy:9:8 : Error: unknown token when expecting an expression
        }
        ^
```

图 5 错误处理实例

2. 表达式的解析

表达式的解析流程比较复杂，因为涉及到了运算符优先级的问题，这带来了多次递归调用，导致一个简单的表达式往往具有复杂的解析流程。

回顾之前的语法树结点，表达式的结点如图 6 所示：

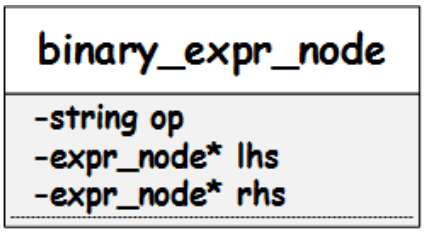

图 6 语法树表达式结点

表达式结点拥有两个子结点，这两个子结点可以指向所有类型的语法树结点，因为所有的语法树结点都继承自 expr_node。该结点也可以用作一元表达式，当用作一元表达式时，rhs 的值应为空。

一次典型的表达式解析流程如图 7 所示，数字表示解析的顺序：

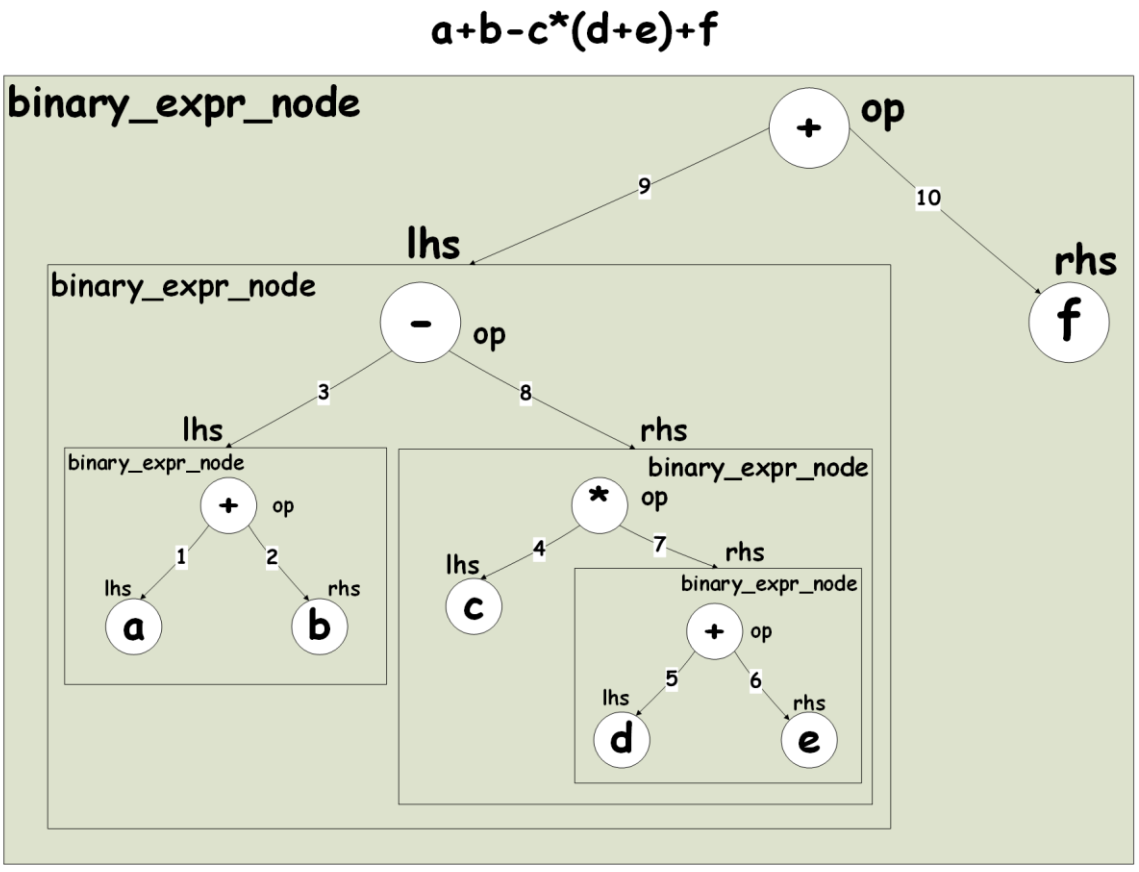

图 7 a+b-c*(d+e)+f 的解析过程

表达式的解析从 parse_expr()函数开始：

```
expr_node* parser::parse_expr() {
    PRINT("parse_expr");
    expr_node* lhs = parse_primary();
    if (!lhs) return 0;

    return parse_bin_op_rhs(0, lhs);
}
```

图 8 parse_expr()函数

该函数调用 parse_primary()函数，解析表达式的左部，若表达式的左部为空，则直接返回 0，若不为空，则再去解析表达式的右部。

众所周知，表达式的左部是一个递归的概念，表达式的左部与右部依然有可能是表达式。这个在 parse_primary()函数中有所体现。

```
expr_node* parser::parse_primary() {
    PRINT("parse_primary");
    c_ttype cur_type = CUR_TOKEN_TYPE;
    switch (cur_type) {
        default: return Error("unknown token when expecting an expression");
        case C_NAME: return parse_identifier_node();
        case C_NUMBER : return parse_number_node();
        case C_OPEN_PAREN : return parse_paren_node();
    }
}
```

图 9 parse_primary()函数

从图 9 中可以看到一个 switch-case 语句，主要内容如下：

- 若当前词法单元为变量名，则直接构造一个变量结点
- 若当前词法单元为常数，则直接构造一个常数结点
- 若当前词法单元为左括号，则调用 parse_paren_node()函数
- 否则直接报错

parse_paren_node()函数非常简单，主要是调用 parse_expr 得到括号内的表达式，最后检测括号是否被正确闭合。

```
expr_node* parser::parse_paren_node() {
    PRINT("parse_paren_node");
    get_next_token(); //skip '('
    expr_node* node = parse_expr();
    if (!node) return 0;

    if (CUR_TOKEN_TYPE != C_CLOSE_PAREN)
        return Error("expected ')'");
    get_next_token();
    return node;
}
```

图 10 parse_paren_node()函数

以上三个函数可以让表达式解析进入到最关键的函数，parse_bin_op_rhs()函数。

```
expr_node* parser::parse_bin_op_rhs(int prev_type, expr_node* lhs) {
    PRINT("parse_bin_op_rhs");
    while (1) {
        int cur_precedence = get_op_precedence();

        if (cur_precedence < prev_type)
            return lhs;

        std::string op = CUR_TOKEN_NAME;
        get_next_token(); // skip binop

        expr_node* rhs = parse_primary();
        if (!rhs)   return 0;

        int next_precedence = get_op_precedence();
        if (cur_precedence < next_precedence) {
            rhs = parse_bin_op_rhs(cur_precedence + 1, rhs);
            if (rhs == 0)   return 0;
        }

        lhs = new binary_expr_node(op, lhs, rhs);
    }
}
```

图 11 parse_bin_op_rhs()函数

该函数主要做了一件事情：检查运算符合法性与优先级，从而决定是合并当前结点还是递归调用自身。

当当前运算符优先级小于下一运算符的优先级时，不能合并当前结点，而应当递归调用自身，完整地解析表达式。一个很明显的例子，**a+b*c;**，当解析到 b 时，已解析的表达式左部(lhs)为 a，当前运算符为加号，b*c 作为待解析表达式，下一运算符为乘号。此时当前运算符结点(加号)优先级小于下一运算符结点(乘号)优先级，如果此时合并结点，那么 a 和 b 就将被合并，表达式将被曲解为(a+b)*c，明显不符合原意。此时正确的选择应当是递归调用自身，再下一次递归时，当前运算符是乘号，而下一运算符时分号，乘号的优先级大于分号，故 b 和 c 此时合成新结点并返回。然后该新结点再与之前的 a 结点合成新结点，最后返回，这就是正确的处理结果。

3. if-else 语句的解析

有了表达式解析函数之后，接下来的 if-else 语句解析就比较方便了，只需解决其中几个关键字的处理即可。在该语法分析器中，使用一个名为 parse_if_statement()的函数来解析。

```
expr_node* parser::parse_if_statement() {
    get_next_token(); // skip if
    if (CUR_TOKEN_TYPE != C_OPEN_PAREN) {
        return Error("expected '(' after if ");
    }

    get_next_token(); // skip '('
    expr_node* cond = parse_expr();
    if (CUR_TOKEN_TYPE != C_CLOSE_PAREN) {
        return Error("expected ')' after if-condition");
    }
    get_next_token(); // skip ')'
```

图 12 parse_if_statement()函数(前半部分)

图 12 是检查 if 语句中条件的部分，只需检查括号并正确跳过括号，如果有错误则报错并返回，正确则保存条件语句并继续解析。

```
    if (CUR_TOKEN_TYPE != C_OPEN_BRACE) {
        return Error("expected '{' before if-statement");
    }
    get_next_token(); // skip '{'

    std::vector<expr_node*> if_stmts;
    while (CUR_TOKEN_TYPE != C_CLOSE_BRACE) {
        expr_node* stmt = parse_expr();
        if_stmts.push_back(stmt);
        get_next_token();
    }

    get_next_token(); // skip '}'

    std::vector<expr_node*> else_stmts;
    if (CUR_TOKEN_TYPE == RID_ELSE) {
        get_next_token(); // skip else
        get_next_token(); // skip '{'
        while (CUR_TOKEN_TYPE != C_CLOSE_BRACE) {
            expr_node* stmt = parse_expr();
            else_stmts.push_back(stmt);
            get_next_token();
        }
    }
    get_next_token(); // skip '}'
    return new if_stmt_node(cond, if_stmts, else_stmts);
}
```

图 13 parse_if_statement()函数(后半部分)

if-else 语句块的处理也是类似的，在处理完之后，返回一个新的 if_stmt_node 结点即可。

3) 中间代码生成的实现

1. 概述

回顾之前提到的，所有语法树结点的共同父节点 expr_node，它要求所有继承它的结点都必须重写 code_gen()函数，从而实现生成中间代码的目的。

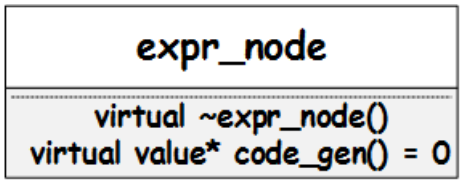

图 14 语法树结点基类

这里以表达式结点与 if-else 结点以及 for 循环结点为例，说明如何重写该函数并生成中间代码。

2. 表达式结点中间代码的生成

如之前所提到的，一个表达式结点可能包含着复杂的结点，所以需要递归地去调用 code_gen()方法。

```
virtual value* code_gen() {
    char tmp_op = op[0];
    value* L = lhs->code_gen();
    value* R = rhs->code_gen();
    if(L == 0||R == 0) return 0;

    switch(tmp_op){
        default:
            std::cout << "Invalid binary operator" << std::endl;
        case '+':
        case '-':
        case '*':
        case '/':
        {
            temp tmp;
            emit(tmp.to_string() + " = " +
            L->get_val() + " " +
            std::string(1, tmp_op) + " " +
            R->get_val());
            return new value(tmp.to_string());
        }
    }
}
```

图 15 二元表达式的 code_gen()函数

此处只支持加减乘除四种运算符，解析时，调用 emit()函数将中间代码输出。输出的结果是 SSA 风格的中间代码，每一个被赋值的变量，都会以一个新的变量来取代。比如 x = a + b * c / (d + 5) - k;会生成如图 16 所示的中间代码：

```
pseudo-code:
    t0 = b * c
    t1 = d + 5
    t2 = t0 / t1
    t3 = a + t2
    t4 = t3 - k
    x = t4
```

图 16 生成的 SSA 风格的中间代码

应当了解的是，该函数并非像表面上看起来那样会陷入无休止的递归，因为 code_gen()是一个虚函数，lhs 与 rhs 可能是众多语法树结点中的一种。当 lhs 与 rhs 是变量结点或常量结点时，调用 code_gen()只是简单地返回一个 value 值而已。

3. if-else 结点中间代码的生成

有了表达式结点代码生成的函数之后，if-else 结点中间代码的生成也变得非常简单。主要是在生成的表达式中间代码之间插入正确的跳转语句。

```
virtual value* code_gen() {
    label l0;
    label l1;
    label l2;
    std::cout << "\tif ";
    cond -> code_gen();
    emit("\tgoto " + l0.to_string());
    emit("goto " + l1.to_string());
    std::cout << l0.to_string() << ": ";
    for (expr_node* e : if_stmts) {
        e -> code_gen();
    }
    emit("goto " + l2.to_string());
    std::cout << l1.to_string() << ": ";
    for (expr_node* e : else_stmts) {
        e -> code_gen();
    }
    std::cout << l2.to_string() << ": ";
    return nullptr;
}
```

图 17 if-else 结点的 code_gen()函数

一个完整的 if-else 语句共由 3 个 label 组成，label0 代表 if 语句块，label1 代表 else 语句块，label2 代表后续代码的开始。该函数的逻辑为：

- 打印 if 条件语句，若满足则 goto 到以 label0 为开头的 if 语句块
- 打印无条件跳转到以 label1 为开头的 else 语句块的语句
- 在 label0 中打印 if 语句块
- 在 label1 中打印 else 语句块
- 打印 label2

一个完整实例如下：

```
Time:07:46:48     User:james     Dir:/mnt/hgfs/For
$./compiler
Lexer Test: 97/97 (100.00%) passed
    source code:
        if (c == 10) {
            a = a + b;
        } else {
            b = a * b;
        }

    pseudo-code:
L0:     if      c == 10
                goto L1
        goto L2
L1:     t0 = a + b
        a = t0
        goto L3
L2:     t1 = a * b
        b = t1
L3:
```

图 18 if-else 结点生成中间代码实例

4. 一个完整的解析结果

下图是一个完整的解析结果，包含了复杂的算术表达式，if-else 语句以及 for 循环语句。

```
Time:07:22:34     User:james     Dir:/mnt/hgfs/For-Linux/LambCompiler/bin
$./compiler
Lexer Test: 97/97 (100.00%) passed
    source code:
        x = a + b * c / (d + 5) - k;
        if (c == 10) {
            a = a + b;
        } else {
            b = a * b;
        }
        for(i = 0; i == 5; i = i + 1) {
            a = a + i;
        }
        first = 5;

    pseudo-code:
L0:     t0 = b * c
        t1 = d + 5
        t2 = t0 / t1
        t3 = a + t2
        t4 = t3 - k
        x = t4
        if      c == 10
                goto L1
        goto L2
L1:     t5 = a + b
        a = t5
        goto L3
L2:     t6 = a * b
        b = t6
L3:     i = 0
L4:     if      i == 5
                goto L5
        goto L6
L5:     t7 = a + i
        a = t7
        t8 = i + 1
        i = t8
        goto L4
L6:     first = 5
```

图 19 中间代码生成的完整例子

四、 实验总结

1. 心得

该实验的难点主要有以下两点：

- 表达式的解析
- 中间代码生成的解决方案

表达式的解析难在递归条件的判定，该条件的判断主要依赖于一张运算符优先级表，如图 20 所示：

```
std::map<c_ttype, int> bin_op_precedence;
bin_op_precedence[C_EQ] = 5;        /*  =  */
bin_op_precedence[C_EQ_EQ] = 15;    /*  == */
bin_op_precedence[C_LESS] = 20;     /*  <  */
bin_op_precedence[C_PLUS] = 35;     /*  +  */
bin_op_precedence[C_MINUS] = 35;    /*  -  */
bin_op_precedence[C_MULT] = 40;     /*  *  */
bin_op_precedence[C_DIV] = 40;      /*  /  */
```

图 20 运算符优先级表

中间代码生成的解决方案难在思维方式的转变，实际上编译器方面的技术已经处于很成熟的阶段，比如中间代码生成的这部分，使用访问者模式是一个较为妥善的方法。最开始我在这一步上想了很久，主要是思维方式的局限。我希望去遍历语法树，并且根据得到的当前结点去生成代码，但是是由调用者来生成的，就像是有一个巨大的 switch-case 语句，根据当前结点的不同来生成不同的代码，但是这样做是非常繁琐的，会导致代码变得极度复杂，耦合性极大。实际上，代码的生成不应该由外部调用函数来负责，应当由每个结点自身来负责，每产生一个结点时，就调用该结点的 code_gen()函数，使其生成对应的中间代码。

2. 拓展

2.1 类型推导与类型检查

从之前的内容中可以看出，测试代码中的变量都是未经声明直接使用的，同时，该语法分析器也不具备类型推导功能，故在该语法分析器中，所有变量的类型都是未知的，所以也不存在类型检查。未来可以考虑加入类型检查以及类型推导功能。

2.2 代码的执行

最终，该编译器生成的目标代码应该运行在虚拟机上，中间代码到目标代码的转换尚未完成，但是虚拟机的部分已经完成了一个小型 demo。

该虚拟机的指令十分简单，具体如图 21 所示：

```
enum assm_op{
    ADD, SUB, MUL, DIV, // 0-3
    PUSH, POP, // 4-5
    JMP, // 6
    MOV, LEA, // 7-8
    LOADA, //9 - move the next number to eax
    LOADB, // 10 - move the next number to ebx
    HALT, // 11
};
```

图 21 虚拟机指令

其具体含义如下图所示：

操作	操作含义
ADD	将eax与ebx寄存器中的值相加， 并将结果存储在eax中
SUB	将eax与ebx寄存器中的值相减， 并将结果存储在eax中
MUL	将eax与ebx寄存器中的值相乘， 并将结果存储在eax中
DIV	将eax与ebx寄存器中的值相除， 并将结果存储在eax中
PUSH a	将a的值压入栈
POP a	将栈顶元素弹到a中
JMP a	无条件跳转到a
MOV a, b	将[b]的值移到操作数[a]中
LEA a, b	将b的值放到a中
LOADA	将栈顶元素弹入eax
LOADB	将栈顶元素弹入ebx
HALT	停机

图 22 虚拟机指令含义说明

此时可以用一个小例子测试该虚拟机的正确性，如图 23 所示：

```
void test() {
    size_t i = 0;
    text[i++] = LOADA; // LOAD 20 to eax
    text[i++] = LOADB; // LOAD 10 to ebx
    text[i++] = ADD;   // ADD eax and ebx
    text[i++] = HALT;  // HALT
    *esp++ = 10;
    *esp++ = 20;
}
```

图 23 虚拟机测试代码及栈帧信息

text 代表代码段，esp 代表栈顶指针，该虚拟机的栈增长方向是从低地址向高地址增长的。执行上述代码，虚拟机会将 20 放入 eax 中，10 放入 ebx 中，然后调用 ADD 指令将两个寄存器中的值相加，最后将值存入 eax，可以得到最终的返回值是 30，说明该虚拟机通过了该简单测试。

未来可以在该虚拟机上加入更多指令，并将编译器生成的代码在虚拟机上运行。