辅助代码部分

一、 简介

与作业同时给出的，包括有若干的已经以 C++编写好的辅助包，这将大大减少同学们一些无关的繁杂的代码编写工作。同学们可能需要或者不需要这些包的辅助。

二、 内容

2.1链表

list.h 内提供了一个泛型实现的链表类。

2.2 字符串表

编译器通常需要处理大量的字符串，例如数字常量、字符串常量、布尔常量等。而且，常常有一些字符串字面值是一致的，所以为了高效的处理这个问题，就引入了字符串表。stringtab.h 和 stringtab_functions.h 中提供了字符串表的实现。

对于一个字符串表，其中的每一项是一个 Entry，每一个 Entry 存储了字符串，字符串的长度以及一个唯一的 int 下标。

我们预先定义好了 4 个字符串表，即 inttable、stringtable、floattable 和 idtable。对应的，每个表的表项为类型 IntEntry、StrEntry、FloatEntry 以及 IdEntry。前三个顾名思义，第四个存储变量名、函数名等字符串。他们都由 Entry 类派生而来。

由于字符串表只对每一个字符串保存一份，所以对内容的比较可以直接比较指针 x==y，但是对于不同类型的 Entry 比较是没有意义的。

字符串表的操作函数，包括了 add_string(char *s, int m)，能够给表上增加一个字符串 s，至多 m 个字符。add_string(char *s)能够添加字符串 s 到表。add_int(long i)能够给字符串表中添加一个 64 位整数，并将其转化为字符串形式。get_string 函数能够获取该项的字符串值。

2.3 符号表

symtab.h 提供了一个非常漂亮的符号表实现，用例方法可以在 symtab_example.cc 中查阅到。

通常，编译器必须要能够处理符号名（变量、函数）及其作用域，提供的符号表代码包就是处理这一情况的，符号表提供了添加符号（addid）、进入子域（enterscope）、离开子域（exitscope）等各项功能。

一般的，每个符号在使用之前都必须要先声明一遍。举个例子

```
func test() Void {
var x Int; // x1
x = 10;
if x < 100 {
    var y Int;
    var x Int; // x2
```

```
        y = 2;
        x = 3; //位置 1
    }
    //位置 2
}
```

在位置 2 处访问变量 y 会失败，因为 y 的作用域仅限于 if 后的语句块内，而位置 1 处的 x 赋值，将会是对 if 块内的 x1 赋值，而和外侧的 x2 无关，倘若在位置 2 处访问 x 变量，实际山访问的是 x1，因为 x2 在离开了其作用域之后便不再生效了。

2.4 杂项函数

utilities.h 中声明了一些可能用到的杂项函数。详情请看代码。

2.5抽象语法树（AST）

2.5.1 非终结符和构造函数

seal-expr.h、seal-stmt.h、seal-decl.h 及他们的实现文件对应的.cc，构成了 Seal 所使用的抽象语法树代码包（部分成员函数需要同学们自己编写）。抽象语法树代码，提供给同学们代码量最大的包。对于 Seal 中的每个非终结符，都有对应定义的一个语法树节点类，同样有一个构造方法，能够构造一个该类的节点。构造方法在语法分析中非常有用，因为同学们需要对每种语法规则规定一个树节点生成规则，这里就利用到了构造方法。可以在 seal-tree.aps 中查看到定义的所有非终结符及其构造方法，例如

```
Phylum Variable;
```

指明了 Seal 语法手册中的 Variable 非终结符；在构造时，可以使用 variable()，即可以返回一个 Variable 对象。

以 Variable 为例，首先查阅到 Seal 语法手册中的形式定义，$Variable ::= ObjectID\ TYPEID$。所以在语法分析的 seal.y 中，可以利用

```
variable  :  OBJECTID TYPEID {
                $$ = variable($1, $2);
             }
             ;
```

来对匹配到的连续的 OBJECTID TYPEID 终结符，规约为一个 variable，这里$1、$2 即匹配式第一项和第二项。其他关于 bison 的规则，请查阅附带的《flex 与 bison》或者其他相关资料。

另外请注意，可能会遇到许多令人疑惑的、形式相近的符号，例如术语 Variable、seal.y 中定义的非终结符变量 variable、构造函数 variable（与非终结符变量同名，如果你喜欢，可以在 seal.y 中将非终结符的名字修改为不引起疑惑的其他名字）、语法树节点指针类型 Variable，请阅读相关代码及定义，区分好这些名字的意义，将对完成任务大有助益。

2.5.2 AST 链表

我们可以看到大体上，非终结符分为两类，如

```
phylum Variable;
phylum Variables = LIST[Variable];
```

即 LIST 类和正常类。LIST 即链表形式组织的连续的 Variable。对于 Variable，在 seal-decl.h

中将 LIST 类的定义为 Variables，而正常类为 Variable。这也非常容易理解，例如在函数的声明中，对输入参数，便是以逗号隔开的多条 Variable 信息。我们对于 LIST 形式组织的非终结符，提供了对应的操作函数，仍然以 Variables 为例（其他的都类似）

```
Variables nil_Variables();
Variables single_Variables(Variable);
Variables append_Variables(Variables,Variables);
Variable nth(int index);
int      len();
```

分别可以创建一个空的 Variables、由单个 Variable 创建一个 Variables、将两个 Variables 链接在一起、获取 Variables 中第 index 个 Variable、返回 Variables 的长度。

仍然以语法分析为例，考虑

$$CallDecl ::= \textbf{func}\ object([Variable[,Variable]^*])StmtBlock$$

这条语法规则，我们需要对并列的多个 Variable 规约为一个 Variables，则规则可能**至少包括**将一个 variable 转化为 variable_list，也可能有对一个 variable_list 后面加一个新的 variable，这样便可以递归的处理一个到任意个参数的情况，也即

```
variable_list :
        variable   {
            $$ = single_Variables($1);
        }
        |   variable_list ',' variable {
            $$ = append_Variables($1, single_Variables($3));
        }
        ;
```

此外，现在假设有一个叫做 l 的 LIST，如果要对其中的每个成员遍历做操作，通常的方法是采取

```
for(int i = l->first(); l->more(i); i=l->next(i))
{…对 l->nth(i)做操作…}
```

2.5.3 AST 树结构

语法分析中，我们的终极目的是构造一个巨大单一树根的 AST 树，对于每一次的规约，都对应了 AST 树中的一个节点。事实上，对于每个非终结符类，都是继承了 tree.h 中声明的基础类 tree_node。基础类提供了 get_line_number 函数，可以返回该语法结构在原文件中发生在多少行，dump 方法能够打印出来某个 AST 节点及其子节点。

2.5.4 类成员

对于语法分析、语义分析和目标代码生成中，我们对每个非终结符以及语法结构的类定义了大致相同但又稍有差异的类成员及类函数。但是都包括了类的构造函数，如果需要的话，可以在原基础上添加任何新的成员函数以及成员，方便实现一些特定的功能。但请切记，添加了声明之后，一定要有定义实现。

2.5.5 类

下面的表给出了每个类的意义

名字	含义	所在文件
Decl_class	函数或变量声明基类	seal-decl.h
VariableDecl_class	变量声明	seal-decl.h
Variable_class	变量	seal-decl.h

CallDecl_class	函数声明	seal-decl.h
Program_class	AST 根节点	seal-stmt.h
Stmt_class	语句基类	seal-stmt.h
StmtBlock_class	语句块	seal-stmt.h
IfStmt_class	if-then-else 条件语句	seal-stmt.h
WhileStmt_class	while 循环语句	seal-stmt.h
ForStmt_class	for 循环语句	seal-stmt.h
ReturnStmt_class	return 返回语句	seal-stmt.h
ContinueStmt_class	continue 语句	seal-stmt.h
BreakStmt_class	break 语句	seal-stmt.h
Expr_class	Expr 表达式基类	seal-expr.h
Call_class	函数调用	seal-expr.h
Actual_class	函数实参	seal-expr.h
Assign_class	=赋值语句	seal-expr.h
Add_class	+加法语句	seal-expr.h
Minus_class	-减法语句	seal-expr.h
Multi_class	*乘法语句	seal-expr.h
Divide_class	/除法语句	seal-expr.h
Mod_class	%模语句	seal-expr.h
Neg_class	-单目负号运算	seal-expr.h
Lt_class	<小于	seal-expr.h
Le_class	<=小于等于	seal-expr.h
Equ_class	==等于	seal-expr.h
Neq_class	!=不等于	seal-expr.h
Ge_class	>=大于等于	seal-expr.h
Gt_class	>大于	seal-expr.h
And_class	&&条件与	seal-expr.h
Or_class	||条件或	seal-expr.h
Xor_class	^条件异或 及 按位异或（语义分析即目标代码生成阶段按照参与运算的类型决定）	seal-expr.h
Not_class	!非	seal-expr.h
Bitnot_class	~单目按位取反	seal-expr.h
Bitand_class	&按位与	seal-expr.h
Bitor_class	|按位或	seal-expr.h
Const_int_class	整数常量	seal-expr.h
Const_float_class	浮点数常量	seal-expr.h
Const_string_class	字符串常量	seal-expr.h
Const_bool_class	布尔常量	seal-expr.h
Object_class	变量符号	seal-expr.h
No_expr_class	空表达式	seal-expr.h

需要特别强调的是，以_class 结尾的是原本定义的类，而不带_class 的为该类的节点类，注意查看相关头文件内的 typedef 语句。

三、 小贴士

有一些可能会犯到的常见错误。

- AST 树包抽象类的操作失误，这应该是 C++的编程问题，但请注意但不限于类型转换、父子指针等操作，也请注意如果要实例化一个类对象，必须对其每一个声明和继承的抽象函数定义好。
- 对于 AST 节点构造时，切勿使用 NULL 作为参数，例如某个 if 语句缺少 else 部分，不要使用 NULL 作为参数，而请用 nil_<非终结符名>来创建一个“空对象”。
- 判断一个 LIST 类型树节点是否为空是，请不要以 x == nil_Expr()的形式，而请用 len 函数
- 判断一个 Expr 是否为空 Expr 时，请勿使用 x==no_expr()，而采用一个虚函数的方法。
- 祝大家好运。