Kaleidoscope 第一至三章阅读报告

1.1、gettok()如何向调用者传递 token 类别、token 语义值？

gettok()的返回值是整型，不同的值代表不同的 token。在枚举类型 Token 中定义了一些符号常量（称为枚举子？），如果需要返回的 token 是 eof、def、extern、identifier、number 五种之一，则返回它们相应的符号常量。否则，返回下一个输入字符的 ASCII 码。

只有当返回的 token 是 identifier 或 number 时，需要返回语义值。对于 identifier，语义值存储在 string 类型的全局变量 IdentifierStr 中，对于 number，语义值存储在 double 类型的全局变量 NumVal 中。

2.1 解释 ExprAST 里的 virtual 的作用，在继承时的原理（解释 vtable)。

当派生类 Derived 中的某个函数 f 试图覆盖基类 Base 中的同名函数时，可能出现这样的问题：如果用一个指向 Base 类的指针或一个类型为 Base 的引用，去访问 Derived 类中的成员函数 f，则只能访问到 Base 类中的函数 f，没有达到覆盖的目的。虚函数的引入是为了解决这个问题。

虚函数是一种特殊的函数类型。当它被调用时，被解析为在基类和派生类之间继承层数最多的那个版本。这样，通过一个指向基类类型的指针、或一个对基类对象的引用，也可以调用派生类中的成员函数。

例如，class A 有成员函数 f，class B 继承自 A，也有同类型的成员函数 f，class C 继承自 B，也有同类型的成员函数 f，A &ref = c 是一个对 C 类对象 c 的类型为 A 的引用。当 A、B 中的 f 都声明为虚函数时，ref.f 调用的是 A 与 C 之间继承层数最多的 f 的版本，即 c.f。

虚函数的实现方式是这样的：每一个使用了虚函数的类，或继承自一个使用了虚函数的基类的派生类中，都有一张虚函数表（vtable），它是由编译器在编译时生成的。表中的每一项都是一个函数指针，它指向该类可以调用的继承层数最多的那个成员函数。此外，在基类中还有一个指向 vtable 的隐藏指针*__vptr。当类的一个实例被创建时，*__vptr 被自动设为指向那个类的 vtable。*__vptr 可以被派生类继承。

这样，在上面的例子中，类 A、B、C 有各自的 vtable 和*__vptr。由于一个类的 vtable 中存放的函数指针指向该类可以调用的继承层数最多的虚函数版本，因此 A 的 vtable 中有一个指向 A::f 的指针，B 的 vtable 中有一个指向 B::f 的指针，C 的 vtable 中有一个指向 C::f 的指针。它们的*__vptr 指向各自的 vtable。当用 ref 去引用 f 时，由于 f 是虚函数，因此程序首先访问*__vptr。虽然 ref 只能看到 c 中属于类 A 的部分属性，但类 C 的*__vptr 是从 A 中继承来的，因此可以被访问。然后，程序通过*__vptr 找到 c 所属的类 C 的 vtable，在其中查找 f 对应的条目，最终找到 C::f 的地址。

2.2 解释代码里的 <std::unique_ptr> 和为什么要使用它？

unique_ptr 是一种智能指针，当程序离开它指向的对象的作用域时，对象会被自动销毁。两个 unique_ptr 不能指向同一个对象，但两个 unique_ptr 之间可以传递所指的对象。

使用 unique_ptr 的好处是不管是正常退出还是异常退出，均可为处理拥有动态寿命的函数和对象提供额外保护。

2.3 阅读 src/toy.cpp 中的 MainLoop 及其调用的函数。阅读 HandleDefinition 和 HandleTopLevelExpression，忽略 Codegen 部分，说明两者对应的 AST 结构。

HandleDefinition 为函数构建 AST，它的两棵子树分别是函数头的 AST 和函数体中表达式的 AST。

HandleTopLevelExpression 为顶层表达式构建 AST，它默认将该表达式封装在一个名为”__anon_expr”、无参数的函数中，建立该函数的 AST，函数体部分是原表达式的 AST。

2.4 Kaleidoscope 如何在 Lexer 和 Parser 间传递信息？（token、语义值、用什么函数和变量）

在 Parser 中通过调用 getNextToken 函数间接调用 Lexer 中的 gettok 函数，将其返回值（即 token）放在全局变量 static int CurTok 中。number 的语义值放在全局变量 double NumVal 中，identifier 的语义值放在全局变量 string IdentifierStr 中。

2.5 Kaleidoscope 如何处理算符优先级（重点解释 ParseBinOpRHS ）？解释 a*b*c、a*b+c、a+b*c 分别是如何被分析处理的？

首先，设置一张表 BioopPrecedence，将各个二元算符的优先级量化存入表中。通过函数 GetTokPrecedence 可以获取当前在 CurTok 中的算符的优先级。

然后，将一个表达式分割成第一个 primaryexpr 和若干个形如[binop，primaryexpr]的对。在 ParseExpression 函数中，首先调用 ParsePrimary 函数返回第一个 primaryexpr，让 LHS 指向它。由于其后可能跟有二元算符，因此调用 ParseBinOpRHS 函数，并将优先级 0 和 LHS 作为参数传给它。

ParseBinOpRHS 函数可以处理若干个[binop，primaryexpr]对。它接受两个参数，ExprPrec 和 LHS。ExprPrec 用来指示 ParseBinOpRHS 函数应当继续分析的算符的最低优先级，LHS 表示表达式中已经分析过的部分。如果后面已经没有[binop，primaryexpr]对，或者下一个二元算符的优先级低于 ExprPrec，则直接将 LHS 返回，第一种情况下返回的是整棵表达式树，第二种情况下返回的是表达式树的子树。

如果下一个二元算符的优先级高于 ExprPrec，则应当继续分析。接下来的动作是读入下一个二元算符放在 BinOp 中，并调用 ParsePrimary 函数读入

primaryexpr，返回的子树指针为 RHS。至此，下一个[binop，primaryexpr]的内容已经获取。

在这个时候，要决定是为已经分析的部分生成表达式子树，还是继续分析后面可能的[binop，primaryexpr]对。这取决于再下一个算符 x 的优先级与 BinOp 的优先级关系。如果 x 的优先级低于 BinOp，则为已经分析的部分建立表达式树，LHS 和 RHS 分别是它的左右子树。否则，分析过程应当继续。ParseBinOpRHS 通过递归调用自身来实现这一目的，传入的参数分别是 BinOp 的优先级+1，和 RHS，这是因为其后分析的表达式中的二元算符应当具有比 BinOp 更高的优先级，并且在它之前已分析的部分是 RHS 指向的子树。

分析样例：

（1）a * b * c

ParseExpression 函数通过调用 ParsePrimary 读入 a；

ParseExpression 函数调用 ParseBinOpRHS 函数，参数为（0,指向子树 a 的指针）；

在第一次循环中，下一个算符*的优先级高于 0，因此 ParseBinOpRHS 函数读取*，b；

由于再下一个算符*的优先级与当前算符相同，ParseBinOpRHS 建立子树 a*b；

在第二次循环中，由于下一个算符*优先级高于 0，因此读取*，c；

右边已没有[binop，primaryexpr]对，因此建立(a*b)*c 对应的子树；

在第三次循环中，下一个符号的优先级低于 0，因此 ParseBinOpRHS 函数返回指向表达式树(a*b)*c 的指针。

（2）a * b + c

ParseExpression 函数通过调用 ParsePrimary 读入 a；

ParseExpression 函数调用 ParseBinOpRHS 函数，参数为（0,指向子树 a 的指针）；

在第一次循环中，下一个算符*的优先级高于 0，因此 ParseBinOpRHS 函数读取*，b；

由于再下一个算符+的优先级低于当前算符，ParseBinOpRHS 建立子树 a*b；

在第二次循环中，由于下一个算符优先级+高于 0，因此读取+，c；

右边已没有[binop，primaryexpr]对，因此建立(a*b)*c 对应的子树；

在第三次循环中，下一个符号的优先级低于 0，因此 ParseBinOpRHS 函数返回指向表达式树(a*b)+c 的指针。

（3）a + b * c

ParseExpression 函数通过调用 ParsePrimary 读入 a；

ParseExpression 函数调用 ParseBinOpRHS 函数，参数为（0,指向子树 a 的指针）；

在第一次循环中，下一个算符+的优先级高于 0，因此 ParseBinOpRHS 函数读取+，b；

由于再下一个算符*的优先级高于当前算符+，因此 ParseBinOpRHS 函数递归调用自身，参数为（21，指向子树 b 的指针）；

在第二层 ParseBinOpRHS 函数中的第一次循环中，下一个算符*的优先级高于 21，因此读取*，c。右边已没有[binop，primaryexpr]对，因此建立 b*c 对应的子树；

在第二层 ParseBinOpRHS 函数中的第二次循环中，下一个字符的优先级低于

21,因此返回指向子树 b*c 的指针；

在第一层 ParseBinOpRHS 函数中，建立子树 a+(b*c)；

在第一层 ParseBinOpRHS 函数的第二次循环中，下一个字符的优先级低于 0，因此返回指向子树 a+(b*c)的指针；

2.6 解释 Error、ErrorP 的作用，举例说明它们在语法分析中的应用。

LogError、LogErrorP 是用于报告错误的辅助函数。LogError 报告在匹配表达式时检测到的错误，LogErrorP 报告在匹配函数头时检测到的错误。

使用 LogError 的例子：LogError 用于报告匹配 primaryexpr 时遇到非法字符的错误。

```
/// primary
///   ::= identifierexpr
///   ::= numberexpr
///   ::= parenexpr
static std::unique_ptr<ExprAST> ParsePrimary() {
  switch (CurTok) {
  default:
    return LogError("unknown token when expecting an expression");
  case tok_identifier:
    return ParseIdentifierExpr();
  case tok_number:
    return ParseNumberExpr();
  case '(':
    return ParseParenExpr();
  }
}
```

使用 LogErrorP 的例子：LogErrorP 用于报告匹配 prototype 时遇到错误函数名或缺失左、右括号的错误。

```
/// prototype
///   ::= id '(' id* ')'
static std::unique_ptr<PrototypeAST> ParsePrototype() {
  if (CurTok != tok_identifier)
    return LogErrorP("Expected function name in prototype");

  std::string FnName = IdentifierStr;
  getNextToken();

  if (CurTok != '(')
    return LogErrorP("Expected '(' in prototype");

  std::vector<std::string> ArgNames;
  while (getNextToken() == tok_identifier)
    ArgNames.push_back(IdentifierStr);
  if (CurTok != ')')
    return LogErrorP("Expected ')' in prototype");

  // success.
  getNextToken(); // eat ')'.

  return helper::make_unique<PrototypeAST>(FnName, std::move(ArgNames));
}
```

2.7 Kaleidoscope 不支持声明变量和给变量赋值，那么变量的作用是什么？

Kaleidoscope 中的变量如果是函数的形参，则可以在函数调用时被初始化。变量可以参与构成表达式，这些表达式也可以作为函数调用时的实参传递给函数中的变量。

3.1 解释教程 3.2 节中 `Module`、`IRBuilder<>` 的作用。

Module 类代表了 LLVM 程序的顶层结构。它表示源程序的一个翻译单元，或由连接器连接的若干单元的组合。它记录着一个函数列表、一个全局变量列表和一张符号表。此外还提供了一些成员函数如`Module::FunctionListType &getFunctionList()`等。

`IRBuilder` 类方便了指令的插入。它支持将若干指令插入一个 `BasicBlock` 尾部或一条特定指令之前。此外，它还支持对常量的折叠和对命名寄存器的重命名等操作。

3.2 为何使用常量时用的函数名都是 `get` 而不是 `create` ？

因为在 LLVM IR 中常量是唯一的，且是共享的，因此创建常量表达式的中间代码时是获取它而不是创建一个新的常量。

3.3 简要说明声明和定义一个函数的过程

声明函数的过程是，先用 Functiontype::get 方法获取一个函数类型。由于 Kaleidoscope 中变量只有 double 一种类型，因此传给 get 方法的参数告知其函数类型是以 N 个 double 类型变量为参数、以一个 double 类型变量为返回值。由于函数类型与常量类似，在 LLVM 中是唯一的，因此使用”get”方法而不是创建一个新的函数类型。

得到函数类型之后，通过 Function::Create 方法创建一个函数声明的中间代码，传给该方法的参数分别是函数类型、连接方式、函数名和所加入的模块。

最后可选的一步是为函数声明中的每个参数赋一个名字。

定义函数的过程是，首先去本模块的符号表中查找该名字的函数声明是否已存在，如果不存在，则先调用 PrototypeAST::codegen 方法生成函数声明的中间代码。在进行下面的操作之前，要先作判断，保证该函数的函数声明已存在且函数体为空。

接着创建一块 BasicBlock，在其构造函数中指明将其插入到待生成的函数中。通过 IRBuilder::SetInserPoint()方法将指令的插入位置设置为该 BasicBlock 尾部。

将函数参数名记录在 NamedValues 表中以便建立表达式时查询。

然后，通过调用相应表达式的 codegen 方法，为函数体中的表达式生成中间代码。最后，通过 verifyFunction 方法对生成的代码进行一致性检查。

3.4 文中提到了 visitor pattern, 虽然这里没有用到, 但是是一个很重要的设计模式, 请调研后给出解释(字数不作限制).

Visitor 的定义是：Represent an operation to be performed on elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Visitor design pattern 是指在面向对象编程中，将算法与对象分离的一种设计模式。这样可以在无需改变数据对象的结构的情况下，在一个 visitor 类中，为数据对象修改现有的操作或实现新的操作。

Visitor pattern 的实现框架如下：

存在两种类型的对象，一种称为”element”，一种称为”visitor”。在一个 element 对象中，有一种 accept 方法，它接受一个 visitor 对象作为参数。进而，该 element 可以调用 visitor 对象中某个特定的 visit 方法，将自身作为参数传给该方法，来实现对自身数据的某种类型的操作。