设计文档(vsl)

1. **抽象语法树(AST)**

抽象语法树（以下简称为AST）是一门编程语言源码的抽象语法结构的树形表示。各种语言组件，例如表达式、条件控制语句等，都有相 应的AST，并被区分为操作符和操作数。AST并不表示这些代码如何由 语法生成，而是表达了语言组件之间的关系。AST忽略了一些无关紧要 的元素，例如标点符号、定界符（通常是空格、换行）。另外，AST中 的每个元素都会有一些附加的属性，在之后的编译阶段会有一定作用。 例如，源码行号信息就是这样一个属性，在进行语法检查遇到语法错误 时就可以输出错误代码的行号信息（在C++的Clang前端中，位置、行 号、列号等信息以及其他相关属性由SourceManager类的一个对象存储）。 AST的使用集中在语义分析阶段，在这个阶段，编译器会检查程序 和语言元素是否正确使用。此外，在语义分析阶段编译器还会基于AST 生成符号表。完整的树遍历允许验证程序的正确性。在验证正确后， AST还是代码生成的基础。

**准备工作:**

在生成AST之时，我们需要运行词法分析器来得到token。我们即将要解析的语言由表达式、函数定义、函数声明组成，而表达式又有多种类型，包括变量、二元运算符、数值表达式等。

**详细步骤:**

为了定义AST结构，执行以下步骤。

1. **打开\*.cpp文件：**

以下是词法分析器代码，定义了AST。

1. **首先定义一个base类解析表达式：**

class BaseAST { public : virtual ~BaseAST(); };

还需要定义几个派生类来解析每一种类型的表达式。

1. **变量表达式的AST类定义如下：**

class VariableAST : public BaseAST{

std::string Var\_Name; // 定义string对象用作存储变量名

public: VariableAST (std::string &name) : Var\_Name(name) {

变量AST类的含参构造函数由传入构造函数的字符串初始化 };

1. **语言会包含一些数值表达式。数值表达式的AST类定义如下：** class NumericAST : public BaseAST { int numeric\_val; public : NumericAST (intval) :numeric\_val(val) {} };
2. **对于由二元运算组成的表达式，AST类定义如下：** Class BinaryAST : public BaseAST { std::string Bin\_Operator; // 用于存储二元运算符的string对象 BaseAST LHS, RHS; // 用于存储一个二元表达式的LHS和RHS的对象。 // 由于LHS和RHS二元操作可以是任何类型，因此用BaseAST对象存储。 public: BinaryAST (std::string op, BaseAST lhs, BaseAST rhs ) : Bin\_Operator(op), LHS(lhs), RHS(rhs) {} // 初始化二元运算符、二元表达式的LHS和RHS };
3. **用于函数声明的AST类定义如下**： class FunctionDeclAST { std::string Func\_Name; std::vector Arguments; public: FunctionDeclAST(const std::string &name, const std::vector &args) : Func\_Name(name), Arguments(args) {}; };
4. **用于函数定义的AST类定义如下**： class FunctionDefnAST { FunctionDeclAST Func\_Decl; BaseAST Body; public: FunctionDefnAST(FunctionDeclAST proto, BaseAST body) : Func\_Decl(proto), Body(body) {} };
5. **用于函数调用的AST类定义如下**： class FunctionCallAST : public BaseAST { std::string Function\_Callee; std::vector<BaseAST\*> Function\_Arguments; public: FunctionCallAST(const std::string &callee, std::vector<BaseAST\*> &args): Function\_Callee(callee), Function\_Arguments(args) {} };

到这里，AST的基本框架已经基本可用。

1. **语法分析器**

语法分析器（parser）根据语言的语法规则来解析代码，解析阶段决定了输入的代码是否能够根据既定的语法组成token流。在此阶段会构造出一棵解析树，而语法分析器则会定义一些函数来把代码组织成一种被称为AST的数据结构。这次实验的解析器采用了递归下降的解析技术自顶向下解析，并用相互递归的函数构建AST。

详细步骤 :

在语法分析器中定义一些基本的变量来持有上下文信息：

1. **打开\*.cpp文件**
2. **定义持有当前token（来自词法分析器）的静态全局变量：** static int Current\_token;
3. **定义一个函数从词法分析器的输入流获得下一个token，如下：** static void next\_token() { Current\_token = get\_token(); }
4. **下一步需要使用前一节定义的AST数据结构，为解析表达式定义函数**。
5. **定义一个泛型函数，来根据由词法分析器确定的token类型调用 特定解析函数**，如下： static BaseAST\* Base\_Parser() { switch (Current\_token) { default: return 0; case IDENTIFIER\_TOKEN : return identifier\_parser(); case NUMERIC\_TOKEN : return numeric\_parser(); case '(' : return paran\_parser(); } }

**思路**:输入流被词法分析器构建成token流并传递给语法分析器。 Current\_token持有当前处理的token。在这一阶段token的类型是已知 的，并根据其类型来调用相应的解析函数来初始化AST。

1. **code generation to LLVM IR(阅读笔记)**

在开始生成LLVM IR之前，还有一些准备工作要做。首先，给每个AST类添加一个虚函数Codegen（code generation），用于实现代码生成：

/// ExprAST - Base class for all expression nodes.class ExprAST {public:

virtual ~ExprAST() {} virtual Value \*Codegen() = 0;};

/// NumberExprAST - Expression class for numeric literals like "1.0".class NumberExprAST : public ExprAST {

double Val;public:

NumberExprAST(double val) : Val(val) {} virtual Value \*Codegen();};...

每种AST节点的Codegen()方法负责生成该类型AST节点的IR代码及其他必要信息，生成的内容以LLVM Value对象的形式返回。LLVM用“Value”类表示“[静态一次性赋值（SSA，Static Single Assignment）](http://en.wikipedia.org/wiki/Static_single_assignment_form)寄存器”或“SSA值”。SSA值最为突出的特点就在于“固定不变”：SSA值经由对应指令运算得出后便固定下来，直到该指令再次执行之前都不可修改。详情请参考[Static Single Assignment](http://en.wikipedia.org/wiki/Static_single_assignment_form)——这个概念并不难，习惯了就好。

除了在ExprAST类体系中添加虚方法以外，还可以利用[visitor模式](http://en.wikipedia.org/wiki/Visitor_pattern)等其他方法来实现代码生成。再次强调，本教程不拘泥于软件工程实践层面的优劣：就当前需求而言，添加虚函数是最简单的方案。

其次，我们还需要一个“Error”方法，该方法与语法解析器里用到的报错函数类似，用于报告代码生成过程中发生的错误（例如引用了未经声明的参数）：

Value \*ErrorV(const char \*Str) { Error(Str); return 0; }

static Module \*TheModule;static IRBuilder<> Builder(getGlobalContext());static std::map<std::string, Value\*> NamedValues;

上述几个静态变量都是用于完成代码生成的。其中TheModule是LLVM中用于存放代码段中所有函数和全局变量的结构。从某种意义上讲，可以把它当作LLVM IR代码的顶层容器。

Builder是用于简化LLVM指令生成的辅助对象。[IRBuilder](http://llvm.org/doxygen/IRBuilder_8h-source.html)类模板的实例可用于跟踪当前插入指令的位置，同时还带有用于生成新指令的方法。

NamedValues映射表用于记录定义于当前作用域内的变量及与之相对应的LLVM表示（换言之，也就是代码的符号表）。在这一版的Kaleidoscope中，可引用的变量只有函数的参数。因此，在生成函数体的代码时，函数的参数就存放在这张表中。

有了这些，就可以开始进行表达式的代码生成工作了。注意，在生成代码之前必须先设置好Builder对象，指明写入代码的位置。现在，我们姑且假设已经万事俱备，专心生成代码即可。

## 表达式代码生成

为表达式节点生成LLVM代码的过程十分简单明了：连带注释只需区区45行代码便足以搞定全部四种表达式节点。首先是数值常量：

Value \*NumberExprAST::Codegen() {

return ConstantFP::get(getGlobalContext(), APFloat(Val));}

LLVM IR中的数值常量是由ConstantFP类表示的。在其内部，具体数值由APFloat（Arbitrary Precision Float，可用于存储任意精度的浮点数常量）表示。这段代码说白了就是新建并返回了一个ConstantFP对象。值得注意的是，在LLVM IR内部，常量都只有一份，并且是共享的。因此，API往往会采用”foo:get(...)“的形式而不是“new foo(...)”或“foo::Create(...)”。

Value \*VariableExprAST::Codegen() {

// Look this variable up in the function.

Value \*V = NamedValues[Name];

return V ? V : ErrorV("Unknown variable name");}

在LLVM中引用变量也很简单。在简化版的Kaleidoscope中，我们大可假设被引用的变量已经在某处被定义并赋值。实际上，位于NamedValues映射表中的变量只可能是函数的调用参数。这段代码首先确认给定的变量名是否存在于映射表中（如果不存在，就说明引用了未定义的变量）然后返回该变量的值。在后续章节中，我们还会对语言做进一步的扩展，让符号表支持“[循环归纳变量](http://llvm.org/docs/tutorial/LangImpl5.html" \l "for)”和[局部变量](http://llvm.org/docs/tutorial/LangImpl7.html" \l "localvars)。

Value \*BinaryExprAST::Codegen() {

Value \*L = LHS->Codegen();

Value \*R = RHS->Codegen();

if (L == 0 || R == 0) return 0;

switch (Op) {

case '+': return Builder.CreateFAdd(L, R, "addtmp");

case '-': return Builder.CreateFSub(L, R, "subtmp");

case '\*': return Builder.CreateFMul(L, R, "multmp");

case '<':

L = Builder.CreateFCmpULT(L, R, "cmptmp");

// Convert bool 0/1 to double 0.0 or 1.0

return Builder.CreateUIToFP(L, Type::getDoubleTy(getGlobalContext()),

"booltmp");

default: return ErrorV("invalid binary operator");

}}

二元运算符的处理就比较有意思了。其基本思想是递归地生成代码，先处理表达式的左侧，再处理表达式的右侧，最后计算整个二元表达式的值。上述代码就opcode的取值用了一个简单的switch语句，从而为各种二元运算符创建出相应的LLVM指令。

在上面的例子中，LLVM的Builder类逐渐开始凸显出自身的价值。你只需想清楚该用哪些操作数（即此处的L和R）生成哪条指令（通过调用CreateFAdd等方法）即可，至于新指令该插入到什么位置，交给IRBuilder就可以了。此外，如果需要，你还可以给生成的指令指定一个名字。

LLVM的优点之一在于此处的指令名只是一个提示。举个例子，假设上述代码生成了多条“addtmp”指令，LLVM会自动给每条指令的名字追加一个自增的唯一数字后缀。指令的local value name完全是可选的，但它能大大提升dump出来的IR代码的可读性。

[LLVM指令](http://llvm.org/docs/LangRef.html" \l "instref)遵循严格的约束：例如，[add指令](http://llvm.org/docs/LangRef.html" \l "i_add)的Left、Right操作数必须同属一个类型，结果的类型则必须与操作数的类型相容。由于Kaleidoscope中的值都是双精度浮点数，add、sub和mul指令的代码得以大大简化。

然而，LLVM要求[fcmp指令](http://llvm.org/docs/LangRef.html" \l "i_fcmp)的返回值类型必须是‘i1’（单比特整数）。问题在于Kaleidoscope只能接受0.0或1.0。为了弥合语义上的差异，我们给fcmp指令配上一条[uitofp指令](http://llvm.org/docs/LangRef.html" \l "i_uitofp)。这条指令会将输入的整数视作无符号数，并将之转换成浮点数。相应地，如果用的是[sitofp指令](http://llvm.org/docs/LangRef.html" \l "i_sitofp)，Kaleidoscope的‘<’运算符将视输入的不同而返回0.0或-1.0。

Value \*CallExprAST::Codegen() {

// Look up the name in the global module table.

Function \*CalleeF = TheModule->getFunction(Callee);

if (CalleeF == 0)

return ErrorV("Unknown function referenced");

// If argument mismatch error.

if (CalleeF->arg\_size() != Args.size())

return ErrorV("Incorrect # arguments passed");

std::vector<Value\*> ArgsV;

for (unsigned i = 0, e = Args.size(); i != e; ++i) {

ArgsV.push\_back(Args[i]->Codegen());

if (ArgsV.back() == 0) return 0;

}

return Builder.CreateCall(CalleeF, ArgsV, "calltmp");}

函数调用的代码生成非常直截了当。上述代码开头的几行是在LLVM Module的符号表中查找函数名。如前文所述，LLVM Module是个容器，待处理的函数全都在里面。只要保证各函数的名字与用户指定的函数名一致，我们就可以利用LLVM的符号表替我们完成函数名的解析。

拿到待调用的函数之后，就递归地生成传入的各个参数的代码，并创建一条LLVM [call指令](http://llvm.org/docs/LangRef.html" \l "i_call)。注意，LLVM默认采用本地的C调用规范，这样以来，就可以毫不费力地调用标准库中的“sin”、“cos”等函数了。

Kaleidoscope中的四种基本表达式的代码生成就介绍完了。尽情地添枝加叶去吧。去试试[LLVM语言参考](http://llvm.org/docs/LangRef.html)上的各种千奇百怪的指令，以当前的基本框架为基础，支持这些指令易如反掌。

## 函数的代码生成

函数原型和函数的代码生成比较繁琐，相关代码不及表达式的代码生成来得优雅，不过却刚好可以用于演示一些重要概念。首先，我们来看看函数原型的代码生成过程：函数定义和外部函数声明都依赖于它。这部分代码一开始是这样的：

Function \*PrototypeAST::Codegen() {

// Make the function type: double(double,double) etc.

std::vector<Type\*> Doubles(Args.size(),

Type::getDoubleTy(getGlobalContext()));

FunctionType \*FT = FunctionType::get(Type::getDoubleTy(getGlobalContext()),

Doubles, false);

Function \*F = Function::Create(FT, Function::ExternalLinkage, Name, TheModule);

短短几行暗藏玄机。首先需要注意的是该函数的返回值类型是“Function\*”而不是“Value\*”。“函数原型”描述的是函数的对外接口（而不是某表达式计算出的值），返回代码生成过程中与之相对应的LLVM Function自然也合情合理。

FunctionType::get调用用于为给定的函数原型创建对应的FunctionType对象。在Kaleidoscope中，函数的参数全部都是double，因此第一行创建了一个包含“N”个LLVM double的vector。随后，FunctionType::get方法以这“N”个double为参数类型、以单个double为返回值类型，创建出一个参数个数不可变（最后一个参数false就是这个意思）的函数类型。注意，和常数一样，LLVM中的类型对象也是单例，应该用“get”而不是“new”来获取。

最后一行实际上创建的是与该函数原型相对应的函数。其中包含了类型、链接方式和函数名等信息，还指定了该函数待插入的模块。“[ExternalLinkage](http://llvm.org/docs/LangRef.html" \l "linkage)”表示该函数可能定义于当前模块之外，且/或可以被当前模块之外的函数调用。Name是用户指定的函数名：如上述代码中的调用所示，既然将函数定义在“TheModule”内，函数名自然也注册在“TheModule”的符号表内。

// If F conflicted, there was already something named 'Name'. If it has a

// body, don't allow redefinition or reextern.

if (F->getName() != Name) {

// Delete the one we just made and get the existing one.

F->eraseFromParent();

F = TheModule->getFunction(Name);

在处理名称冲突时，Module的符号表与Function的符号表类似：在模块中添加新函数时，如果发现函数名与符号表中现有的名称重复，新函数会被默默地重命名。上述代码用于检测函数有否被定义过。

对于Kaleidoscope，在两种情况下允许重定义函数：第一，允许对同一个函数进行多次extern声明，前提是所有声明中的函数原型保持一致（由于只有一种参数类型，我们只需要检查参数的个数是否匹配即可）。第二，允许先对函数进行extern声明，再定义函数体。这样一来，才能定义出相互递归调用的函数。

为了实现这些功能，上述代码首先检查是否存在函数名冲突。如果存在，（调用eraseFunctionParent）将刚刚创建的函数对象删除，然后调用getFunction获取与函数名相对应的函数对象。请注意，LLVM中有很多erase形式和remove形式的API。remove形式的API只会将对象从父对象处摘除并返回。erase形式的API不仅会摘除对象，还会将之删除。

// If F already has a body, reject this.

if (!F->empty()) {

ErrorF("redefinition of function");

return 0;

}

// If F took a different number of args, reject.

if (F->arg\_size() != Args.size()) {

ErrorF("redefinition of function with different # args");

return 0;

}

为了在上述代码的基础上进一步进行校验，我们来看看之前定义的函数对象是否为“空”。换言之，也就是看看该函数有没有定义基本块。没有基本块就意味着该函数尚未定义函数体，只是一个前导声明。如果已经定义了函数体，就不能继续下去了，抛出错误予以拒绝。如果之前的函数对象只是个“extern”声明，则检查该函数的参数个数是否与当前的参数个数相符。如果不符，抛出错误。

// Set names for all arguments.

unsigned Idx = 0;

for (Function::arg\_iterator AI = F->arg\_begin(); Idx != Args.size();

++AI, ++Idx) {

AI->setName(Args[Idx]);

// Add arguments to variable symbol table.

NamedValues[Args[Idx]] = AI;

}

最后，遍历函数原型的所有参数，为这些LLVM Argument对象逐一设置参数名，并将这些参数注册倒NamedValues映射表内，以备AST节点类VariableExprAST稍后使用。完事之后，将Function对象返回。注意，此处并不检查参数名冲突与否（说的是“extern foo(a b a”这样的情况）。按照之前的讲解，要加上这一重检查易如反掌。

Function \*FunctionAST::Codegen() {

NamedValues.clear();

Function \*TheFunction = Proto->Codegen();

if (TheFunction == 0)

return 0;

下面是函数定义的代码生成过程，开场白很简单：生成函数原型（Proto）的代码并进行校验。与此同时，需要清空NamedValues映射表，确保其中不会残留之前代码生成过程中的产生的内容。函数原型的代码生成完毕后，一个现成的LLVM Function对象就到手了。

// Create a new basic block to start insertion into.

BasicBlock \*BB = BasicBlock::Create(getGlobalContext(), "entry", TheFunction);

Builder.SetInsertPoint(BB);

if (Value \*RetVal = Body->Codegen()) {

现在该开始设置Builder对象了。第一行新建了一个名为“entry”的[基本块](http://en.wikipedia.org/wiki/Basic_block)对象，稍后该对象将被插入TheFunction。第二行告诉Builder，后续的新指令应该插至刚刚新建的基本块的末尾处。LLVM基本块是用于定义[控制流图（Control Flow Graph）](http://en.wikipedia.org/wiki/Control_flow_graph)的重要部件。当前我们还不涉及到控制流，所以所有的函数都只有一个基本块。这个问题我们留到[第五章](https://llvm-tutorial-cn.readthedocs.io/en/latest/chapter-5.html" \l "chapter-5)再改 :-)

if (Value \*RetVal = Body->Codegen()) {

// Finish off the function.

Builder.CreateRet(RetVal);

// Validate the generated code, checking for consistency.

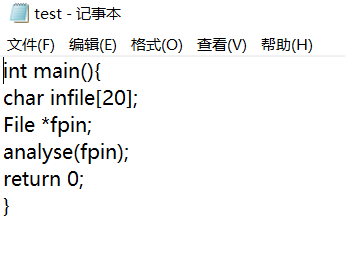
verifyFunction(\*TheFunction);

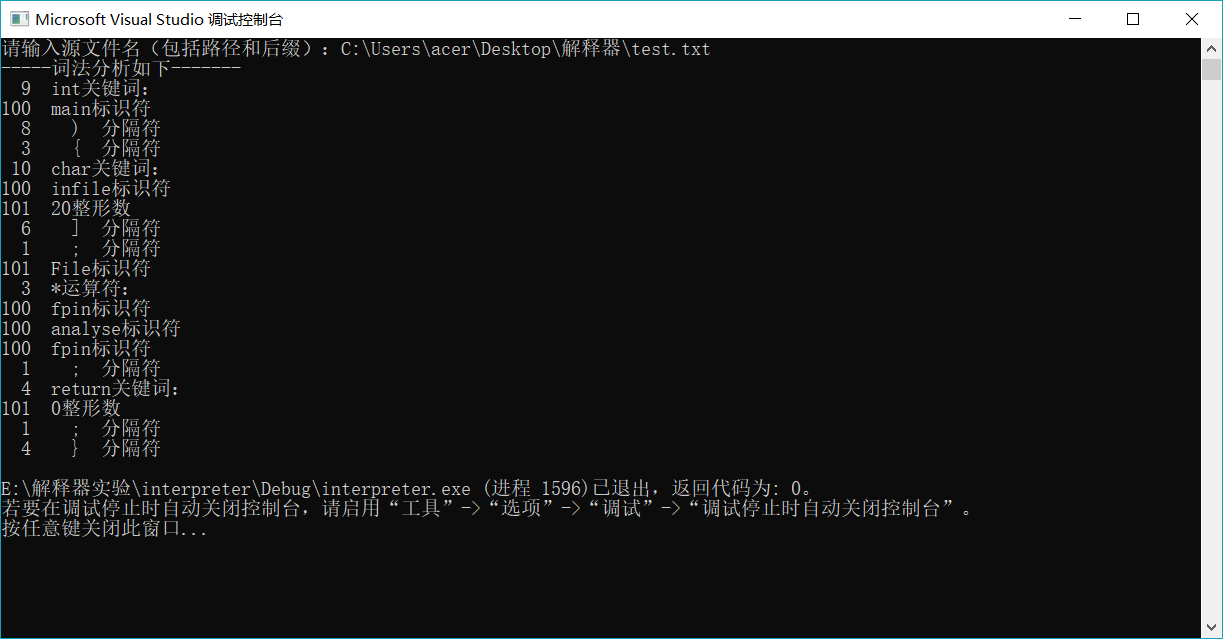
return TheFunction;

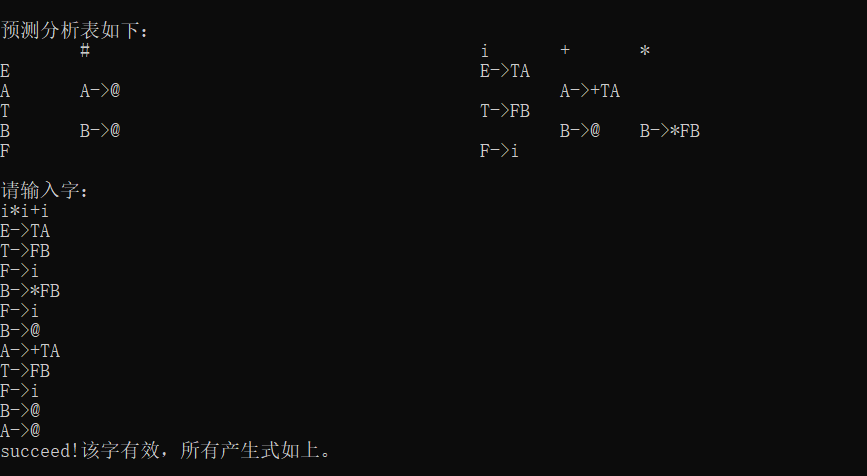
}

选好插入点后，调用函数主表达式的CodeGen()方法即可.

实验输出情况：







小组成员：

2016302580021 阿依多斯

2016302580062 周彪