**第五章阅读笔记**

这章主要扩展Kaleidoscope，使其具有if / then / else表达式和一个简单的'for'循环。

1. if/then/else的lexer扩展

词法分析器扩展很简单。首先，我们为相关标记添加新的枚举值：

***// controltok\_if = -6,tok\_then = -7,tok\_else = -8,***

一旦我们有了这个，我们就会识别词法分析器中的新关键字。这很简单：

***...if (IdentifierStr == "def")***

***return tok\_def;if (IdentifierStr == "extern")***

***return tok\_extern;if (IdentifierStr == "if")***

***return tok\_if;if (IdentifierStr == "then")***

***return tok\_then;if (IdentifierStr == "else")***

***return tok\_else;return tok\_identifier;***

1. if/then/else的AST扩展

为了表示新表达式，我们为它添加了一个新的AST节点：

***/// IfExprAST - Expression class for if/then/else.***

***class IfExprAST : public ExprAST {***

***std::unique\_ptr<ExprAST> Cond, Then, Else;***

***public:***

***IfExprAST(std::unique\_ptr<ExprAST> Cond, std::unique\_ptr<ExprAST> Then,***

***std::unique\_ptr<ExprAST> Else)***

***: Cond(std::move(Cond)), Then(std::move(Then)), Else(std::move(Else)) {}***

***Value \*codegen() override;};***

AST节点只有指向各种子表达式的指针。

3.if/else/then的解析扩展

现在我们有来自词法分析器的相关标记，并且我们要构建AST节点，我们的解析逻辑相对简单。首先我们定义一个新的解析函数：

***/// ifexpr ::= 'if' expression 'then' expression 'else' expressionstatic std::unique\_ptr<ExprAST> ParseIfExpr() {***

***getNextToken(); // eat the if.***

***// condition.***

***auto Cond = ParseExpression();***

***if (!Cond)***

***return nullptr;***

***if (CurTok != tok\_then)***

***return LogError("expected then");***

***getNextToken(); // eat the then***

***auto Then = ParseExpression();***

***if (!Then)***

***return nullptr;***

***if (CurTok != tok\_else)***

***return LogError("expected else");***

***getNextToken();***

***auto Else = ParseExpression();***

***if (!Else)***

***return nullptr;***

***return llvm::make\_unique<IfExprAST>(std::move(Cond), std::move(Then),***

***std::move(Else));}***

接下来我们将它作为主要表达式连接起来：

***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();***

***case tok\_if:***

***return ParseIfExpr();***

}}

4.用于if/else/then的IR.

现在我们已经解析并构建了AST，最后一部分是添加LLVM代码生成支持。这是if / then / else示例中最有趣的部分，因为这是它开始引入新概念的地方。上面的所有代码都已在前面的章节中详细介绍过。

为了激发我们想要生成的代码，让我们看一个简单的例子。考虑：

***extern foo();extern bar();def baz(x) if x then foo() else bar();***

如果您禁用优化，您（很快）从Kaleidoscope获取的代码如下所示：

***declare double @foo()***

***declare double @bar()***

***define double @baz(double %x) {entry:***

***%ifcond = fcmp one double %x, 0.000000e+00***

***br i1 %ifcond, label %then, label %else***

***then: ; preds = %entry***

***%calltmp = call double @foo()***

***br label %ifcont***

***else: ; preds = %entry***

***%calltmp1 = call double @bar()***

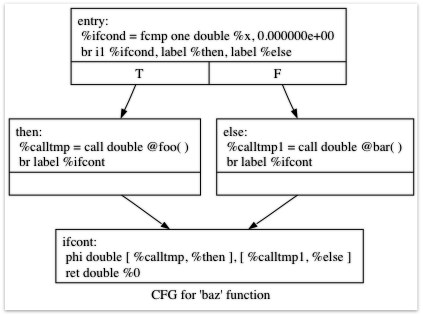
***br label %ifcont***

***ifcont: ; preds = %else, %then***

***%iftmp = phi double [ %calltmp, %then ], [ %calltmp1, %else ]***

***ret double %iftmp}***

要可视化控制流图，您可以使用LLVM“ [opt](http://llvm.org/cmds/opt.html) ”工具的一个漂亮功能。如果你把这个LLVM IR放到“t.ll”并运行“ ”，[会弹出一个窗口](http://llvm.org/docs/ProgrammersManual.html" \l "viewing-graphs-while-debugging-code)，你会看到这个图：llvm-as < t.ll | opt -analyze -view-cfg



另一种方法是通过在代码中插入实际调用并重新编译或在调试器中调用它们来调用“ F->viewCFG()”或“ F->viewCFGOnly()”（其中F是“ Function\*”）。LLVM具有许多用于可视化各种图形的很好的功能。

回到生成的代码，它非常简单：条目块评估条件表达式（在我们的例子中是“x”），并将结果与​​“ ”指令“（”一个“是”有序和不等于“）进行比较。 ）。根据此表达式的结果，代码跳转到“then”或“else”块，其中包含true / false情况的表达式。fcmp one

一旦then / else块完成执行，它们都会回到'ifcont'块以执行在if / then / else之后发生的代码。在这种情况下，唯一要做的就是返回函数的调用者。那么问题就变成了：代码如何知道返回哪个表达式？

这个问题的答案涉及一个重要的SSA操作： [Phi操作](http://en.wikipedia.org/wiki/Static_single_assignment_form)。如果你不熟悉SSA，[维基百科的文章](http://en.wikipedia.org/wiki/Static_single_assignment_form) 是一个很好的介绍，你最喜欢的搜索引擎上有各种其他介绍。简短的版本是Phi操作的“执行”需要“记住”哪个块控制来自。Phi操作采用与输入控制块相对应的值。在这种情况下，如果控制来自“then”块，它将获得“calltmp”的值。如果控制来自“else”块，则它获得“calltmp1”的值。

## [5.'for'循环表达式](http://llvm.org/docs/tutorial/LangImpl05.html" \l "id9)

现在我们知道如何向语言添加基本控制流构造，我们有了添加更多功能的工具。让我们添加更积极的东西，一个'for'表达式：

***extern putchard(char);def printstar(n)***

***for i = 1, i < n, 1.0 in***

***putchard(42); # ascii 42 = '\*'***

***# print 100 '\*' charactersprintstar(100);***

该表达式定义了一个新变量（在这种情况下为“i”），它从起始值迭代，而条件（在这种情况下为“i <n”）为真，递增一个可选步长值（在这种情况下为“1.0”） ）。如果省略步长值，则默认为1.0。循环为true时，它会执行其正文表达式。因为我们没有更好的回报，所以我们只需将循环定义为始终返回0.0。将来当我们有可变变量时，它会变得更有用。

和以前一样，让我们​​来谈谈我们需要Kaleidoscope来支持这一变化。

### [6.'for'循环的Lexer扩展](http://llvm.org/docs/tutorial/LangImpl05.html" \l "id10)

词法分析器扩展与if / then / else相同：

***... in enum Token ...// controltok\_if = -6, tok\_then = -7, tok\_else = -8,tok\_for = -9, tok\_in = -10***

***... in gettok ...if (IdentifierStr == "def")***

***return tok\_def;if (IdentifierStr == "extern")***

***return tok\_extern;if (IdentifierStr == "if")***

***return tok\_if;if (IdentifierStr == "then")***

***return tok\_then;if (IdentifierStr == "else")***

***return tok\_else;if (IdentifierStr == "for")***

***return tok\_for;if (IdentifierStr == "in")***

***return tok\_in;return tok\_identifier;***

### [7.'for'循环的AST扩​​展](http://llvm.org/docs/tutorial/LangImpl05.html" \l "id11)

AST节点也很简单。它基本上归结为捕获节点中的变量名称和组成表达式。

***/// ForExprAST - Expression class for for/in.class ForExprAST : public ExprAST {***

***std::string VarName;***

***std::unique\_ptr<ExprAST> Start, End, Step, Body;***

***public:***

***ForExprAST(const std::string &VarName, std::unique\_ptr<ExprAST> Start,***

***std::unique\_ptr<ExprAST> End, std::unique\_ptr<ExprAST> Step,***

***std::unique\_ptr<ExprAST> Body)***

***: VarName(VarName), Start(std::move(Start)), End(std::move(End)),***

***Step(std::move(Step)), Body(std::move(Body)) {}***

***Value \*codegen() override;};***

### [8.'for'循环的解析器扩展](http://llvm.org/docs/tutorial/LangImpl05.html" \l "id12)

解析器代码也是相当标准的。这里唯一有趣的是处理可选的步骤值。解析器代码通过检查第二个逗号是否存在来处理它。如果不是，则在AST节点中将步长值设置为null：

*/****// forexpr ::= 'for' identifier '=' expr ',' expr (',' expr)? 'in' expressionstatic std::unique\_ptr<ExprAST> ParseForExpr() {***

***getNextToken(); // eat the for.***

***if (CurTok != tok\_identifier)***

***return LogError("expected identifier after for");***

***std::string IdName = IdentifierStr;***

***getNextToken(); // eat identifier.***

***if (CurTok != '=')***

***return LogError("expected '=' after for");***

***getNextToken(); // eat '='.***

***auto Start = ParseExpression();***

***if (!Start)***

***return nullptr;***

***if (CurTok != ',')***

***return LogError("expected ',' after for start value");***

***getNextToken();***

***auto End = ParseExpression();***

***if (!End)***

***return nullptr;***

***// The step value is optional.***

***std::unique\_ptr<ExprAST> Step;***

***if (CurTok == ',') {***

***getNextToken();***

***Step = ParseExpression();***

***if (!Step)***

***return nullptr;***

***}***

***if (CurTok != tok\_in)***

***return LogError("expected 'in' after for");***

***getNextToken(); // eat 'in'.***

***auto Body = ParseExpression();***

***if (!Body)***

***return nullptr;***

***return llvm::make\_unique<ForExprAST>(IdName, std::move(Start),***

***std::move(End), std::move(Step),***

***std::move(Body));}***

我们再次把它作为主要表达方式连接起来：

***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();***

***case tok\_if:***

***return ParseIfExpr();***

***case tok\_for:***

***return ParseForExpr();***

***}}***

### [9.。用于'for'循环的](http://llvm.org/docs/tutorial/LangImpl05.html" \l "id13)IR

现在我们得到了很好的部分：我们想为这件事生成的LLVM IR。通过上面的简单示例，我们获得了这个LLVM IR（请注意，为了清晰起见，生成此转储并禁用优化）：

***declare double @putchard(double)***

***define double @printstar(double %n) {entry:***

***; initial value = 1.0 (inlined into phi)***

***br label %loop***

***loop: ; preds = %loop, %entry***

***%i = phi double [ 1.000000e+00, %entry ], [ %nextvar, %loop ]***

***; body***

***%calltmp = call double @putchard(double 4.200000e+01)***

***; increment***

***%nextvar = fadd double %i, 1.000000e+00***

***; termination test***

***%cmptmp = fcmp ult double %i, %n***

***%booltmp = uitofp i1 %cmptmp to double***

***%loopcond = fcmp one double %booltmp, 0.000000e+00***

***br i1 %loopcond, label %loop, label %afterloop***

***afterloop: ; preds = %loop***

***; loop always returns 0.0***

***ret double 0.000000e+00}***

这个循环包含我们之前看到的所有相同结构：一个phi节点，几个表达式和一些基本块。让我们看看这是如何组合在一起的。

### [10.“for”循环的代码生成](http://llvm.org/docs/tutorial/LangImpl05.html" \l "id14)

codegen的第一部分非常简单：我们只输出循环值的起始表达式：

***Value \*ForExprAST::codegen() {***

***// Emit the start code first, without 'variable' in scope.***

***Value \*StartVal = Start->codegen();***

***if (!StartVal)***

***return nullptr;***

完成此操作后，下一步是为循环体的启动设置LLVM基本块。在上面的例子中，整个循环体是一个块，但请记住，体代码本身可以由多个块组成（例如，如果它包含if / then / else或for / in表达式）。

***// Make the new basic block for the loop header, inserting after current// block.Function \*TheFunction = Builder.GetInsertBlock()->getParent();BasicBlock \*PreheaderBB = Builder.GetInsertBlock();BasicBlock \*LoopBB =***

***BasicBlock::Create(TheContext, "loop", TheFunction);***

***// Insert an explicit fall through from the current block to the LoopBB.Builder.CreateBr(LoopBB);***

这段代码与我们在if / then / else中看到的类似。因为我们需要它来创建Phi节点，所以我们记住了进入循环的块。一旦我们有了这个，我们创建实际的块来启动循环并为两个块之间的连接创建一个无条件分支。

***// Start insertion in LoopBB.Builder.SetInsertPoint(LoopBB);***

***// Start the PHI node with an entry for Start.PHINode \*Variable = Builder.CreatePHI(Type::getDoubleTy(TheContext),***

***2, VarName.c\_str());Variable->addIncoming(StartVal, PreheaderBB);***

现在已经设置了循环的“预读器”，我们切换到循环体的发射代码。首先，我们移动插入点并为循环归纳变量创建PHI节点。由于我们已经知道起始值的传入值，因此我们将其添加到Phi节点。请注意，Phi最终将获得备份的第二个值，但我们还无法设置它（因为它不存在！）。

*/****/ Within the loop, the variable is defined equal to the PHI node. If it// shadows an existing variable, we have to restore it, so save it now.Value \*OldVal = NamedValues[VarName];NamedValues[VarName] = Variable;***

***// Emit the body of the loop. This, like any other expr, can change the// current BB. Note that we ignore the value computed by the body, but don't// allow an error.if (!Body->codegen())***

***return nullptr;***

现在代码开始变得更有趣了。我们的'for'循环为符号表引入了一个新变量。这意味着我们的符号表现在可以包含函数参数或循环变量。为了解决这个问题，在我们编写循环体之前，我们将循环变量添加为其名称的当前值。请注意，外部作用域中可能存在同名变量。很容易使这个错误（发出错误并返回null，如果已经存在VarName的条目）但我们选择允许变量的阴影。为了正确处理这个问题，我们记住了我们可能隐藏的值OldVal（如果没有阴影变量，则为null）。

一旦将循环变量设置到符号表中，代码就会递归代码为body。这允许正文使用循环变量：对它的任何引用自然会在符号表中找到它。

***// Emit the step value.Value \*StepVal = nullptr;if (Step) {***

***StepVal = Step->codegen();***

***if (!StepVal)***

***return nullptr;} else {***

***// If not specified, use 1.0.***

***StepVal = ConstantFP::get(TheContext, APFloat(1.0));}***

***Value \*NextVar = Builder.CreateFAdd(Variable, StepVal, "nextvar");***

现在身体被发射，我们通过添加步长值计算迭代变量的下一个值，如果不存在则计算1.0。' NextVar'将是循环的下一次迭代中循环变量的值。

***// Compute the end condition.Value \*EndCond = End->codegen();if (!EndCond)***

***return nullptr;***

***// Convert condition to a bool by comparing non-equal to 0.0.EndCond = Builder.CreateFCmpONE(***

***EndCond, ConstantFP::get(TheContext, APFloat(0.0)), "loopcond");***

最后，我们评估循环的退出值，以确定循环是否应该退出。这反映了if / then / else语句的条件评估。

***// Create the "after loop" block and insert it.BasicBlock \*LoopEndBB = Builder.GetInsertBlock();BasicBlock \*AfterBB =***

***BasicBlock::Create(TheContext, "afterloop", TheFunction);***

***// Insert the conditional branch into the end of LoopEndBB.Builder.CreateCondBr(EndCond, LoopBB, AfterBB);***

***// Any new code will be inserted in AfterBB.Builder.SetInsertPoint(AfterBB);***

随着循环体的代码完成，我们只需要为它完成控制流程。此代码记住结束块（对于phi节点），然后为循环退出创建块（“afterloop”）。根据退出条件的值，它创建一个条件分支，在再次执行循环和退出循环之间选择。任何将来的代码都会在“afterloop”块中发出，因此它会为其设置插入位置。

***// Add a new entry to the PHI node for the backedge.***

***Variable->addIncoming(NextVar, LoopEndBB);***

***// Restore the unshadowed variable.***

***if (OldVal)***

***NamedValues[VarName] = OldVal;***

***else***

***NamedValues.erase(VarName);***

***// for expr always returns 0.0.***

***return Constant::getNullValue(Type::getDoubleTy(TheContext));}***

最终代码处理各种清理：现在我们有“NextVar”值，我们可以将传入值添加到循环PHI节点。之后，我们从符号表中删除循环变量，使其在for循环后不在范围内。最后，for循环的代码生成总是返回0.0，这就是我们返回的内容 ForExprAST::codegen()。

有了这个，我们总结了本教程的“向Kaleidoscope添加控制流”一章。在本章中，我们添加了两个控制流构造，并使用它们来激发LLVM IR的几个方面，这些方面对于前端实现者来说非常重要。在我们的传奇的下一章中，我们会变得有点疯狂，并为我们可怜的无辜语言添加[用户定义的运算符](http://llvm.org/docs/tutorial/LangImpl06.html)。

小组成员：

2016302580021 阿依多斯

2016302580062 周彪