目 录

[1. 概述 1](#_Toc313350746)

[2. toy 各个example的概述 1](#_Toc2080843016)

[2.1. toy 1的概述 1](#_Toc1715581947)

[2.2. toy 2(发射基本 MLIR)的概述 1](#_Toc1995654388)

[2.2.1. 简介：多级中间表示 1](#_Toc884066587)

[2.2.2. 与 MLIR 接口 2](#_Toc862724430)

[2.2.2.1. 不透明的接口 3](#_Toc1593541675)

[2.2.3. 定义toy方言 3](#_Toc886548523)

[2.2.4. 定义toy操作 4](#_Toc510477811)

[2.2.4.1. Op vs Operation：使用MLIR操作 5](#_Toc1467017264)

[2.2.4.2. 使用操作定义规范（ODS）框架 6](#_Toc109013678)

[2.2.4.2.1. 定义参数和结果 7](#_Toc1154183858)

[2.2.4.2.2. 添加文档 7](#_Toc2129615717)

[2.2.4.2.3. 验证操作语义 7](#_Toc1708498542)

[2.2.4.2.4. 附加build方法  8](#_Toc1293690199)

[2.2.4.2.5. 指定自定义程序集格式  9](#_Toc38800571)

[2.2.5. 完整的玩具示例 11](#_Toc2055321603)

[2.3. 特定于高级语言的分析和转换 12](#_Toc636231309)

[2.3.1. 使用 C++ 样式模式匹配和重写来优化 Transpose  12](#_Toc1269722845)

[2.3.2. 使用 DRR 优化reshapes 14](#_Toc1075261917)

[2.4. 使用接口启用通用转换 15](#_Toc1936850127)

[2.4.1. 背景：应对可扩展的 IR  15](#_Toc315437791)

[2.4.2. Shape Inference：为代码生成做准备 15](#_Toc112766127)

[2.4.2.1. 内联  15](#_Toc717722583)

[2.4.2.2. Intraprocedural shape inference 19](#_Toc1828221957)

[2.5. 部分降级到低级方言的优化 21](#_Toc548889404)

[2.5.1. 方言转换 21](#_Toc1787306633)

[2.5.1.1. 转换目标  22](#_Toc2013990608)

[2.5.1.2. 转换模式  22](#_Toc1998850773)

[2.5.1.3. 部分降低  23](#_Toc519777788)

[2.5.2. 完整的玩具示例  24](#_Toc812237624)

[2.5.3. 利用仿射优化  25](#_Toc164717871)

[2.6. 降低到 LLVM 和代码生成 26](#_Toc453137156)

[2.6.1. 降低到 LLVM  26](#_Toc380335923)

[2.6.1.1. 转换目标  27](#_Toc12888611)

[2.6.1.2. 类型转换器  27](#_Toc1337203743)

[2.6.1.3. 转换模式 28](#_Toc1243060353)

[2.6.1.4. 完全下降  28](#_Toc1606430287)

[2.6.2. CodeGen：摆脱 MLIR  29](#_Toc76268618)

[2.6.2.1. emitting LLVM IR  29](#_Toc1753538164)

[2.6.2.2. 设置 JIT  31](#_Toc925963903)

[2.7. 为玩具添加复合类型 31](#_Toc185282296)

[2.7.1. 在玩具中 定义 struct 32](#_Toc760238375)

[2.7.2. struct在 MLIR 中 定义 32](#_Toc908095972)

[2.7.2.1. 定义类型类  32](#_Toc1893780839)

[2.7.2.1.1. 定义存储类  33](#_Toc2053928574)

[2.7.2.1.2. 定义类型类  34](#_Toc946896543)

[2.7.2.2. Exposing to ODS 35](#_Toc1801618794)

[2.7.3. 解析和打印  35](#_Toc542676235)

[2.7.3.1. 解析 35](#_Toc69135741)

[2.7.3.2. print 36](#_Toc729397063)

[2.7.4. Operating on StructType 37](#_Toc332042714)

[2.7.4.1. 更新现有操作  37](#_Toc384573532)

[2.7.4.2. 添加新Toy操作  37](#_Toc842163191)

[2.7.4.3. 优化操作StructType  38](#_Toc1049765297)

[3. toy语言介绍及其AST的定义 40](#_Toc65311841)

[3.1. 语言 40](#_Toc1391052595)

[3.2. AST 41](#_Toc689588282)

[4. 源码分析 42](#_Toc2079302450)

[4.1. toy1的源码分析 42](#_Toc1242419720)

[4.1.1. 简介 42](#_Toc1209366070)

[4.1.2. parse的处理流程 42](#_Toc744056426)

[4.1.2.1. parse 基础处理 42](#_Toc1407137592)

[4.1.2.2. parsePrototype处理 43](#_Toc1662503226)

[4.1.2.3. parseBlock的处理 44](#_Toc1124392350)

[4.1.2.4. parseReturn 处理 45](#_Toc1420026203)

[4.1.2.5. parseDeclaration处理 46](#_Toc852223321)

[4.1.2.6. parseExpression 处理 46](#_Toc219969055)

[4.1.2.7. parsePrimary处理 47](#_Toc878972842)

[4.1.2.8. parseBinOpRHS 处理 47](#_Toc928491939)

[4.1.2.9. getTokPrecedence处理 48](#_Toc1973507220)

[4.1.2.10. parseParenExpr处理’()’ 48](#_Toc1804936745)

[4.1.2.11. parseTensorLiteralExpr 处理[] 49](#_Toc1113774236)

[4.1.3. Lex的处理流程 49](#_Toc586261947)

[4.1.3.1. Lexer::getCurToken 、Lexer::getNextToken、Lexer::consume 49](#_Toc565549070)

[4.1.3.2. Lexer::getId、Lexer::getValue 49](#_Toc860071427)

[4.1.3.3. Lexer::getLastLocation、Lexer::getLine、Lexer::getCol 49](#_Toc492706873)

[4.1.3.4. Lexer::getTok 从标准输入中获取下一个token 50](#_Toc1512445613)

[4.1.4. AST的处理 50](#_Toc514206573)

[4.1.4.1. XXX::classof 类操作 50](#_Toc1035383108)

[4.1.4.2. ExprAST::getKind获取当前AST类型 50](#_Toc1581581354)

[4.1.4.3. ExprAST::loc 获取当前表达式的起始location 50](#_Toc1243603636)

[4.1.4.4. 子类的操作列表 51](#_Toc1367425822)

[4.2. toy2的源码分析 51](#_Toc1966154887)

[4.2.1. 简介 51](#_Toc2085766827)

[5. 术语 51](#_Toc269707471)

[6. 参考文档 52](#_Toc2031466728)

# 概述

https://mlir.llvm.org/docs/Tutorials/Toy/

本教程介绍了在MLIR上实现基本toy语言的过程。本教程的目标是介绍MLIR的概念；特别是，方言如何帮助轻松支持特定于语言的构造和转换，同时仍然提供了一条通往LLVM或其他代码生成基础设施的捷径。本教程基于[LLVM万花筒教程](https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/index.html)的模型。

另一个很好的介绍来源是2020年LLVM开发会议的在线录音（幻灯片）。

本教程假设您已经克隆并构建了MLIR；如果您还没有这样做，请参阅MLIR入门。

本教程分为以下章节：

第1章：toy语言介绍及其AST的定义。

第2章：遍历AST以在MLIR中发出方言，介绍基本的MLIR概念。在这里，我们展示了如何开始将语义附加到MLIR中的自定义操作。

第3章：使用模式重写系统进行高级语言特定优化。

第4章：使用接口编写通用方言独立转换。在这里，我们将展示如何将特定于方言的信息插入到诸如形状推断和内联之类的通用转换中。

第五章：部分降低到较低水平的方言。我们将把一些高级语言特定的语义转换为面向仿射的通用方言以进行优化。

第6章：降低到LLVM和代码生成。在这里，我们将针对LLVMIR生成代码，并详细介绍降低框架。

第7章：扩展toy：添加对复合类型的支持。我们将演示如何向MLIR添加自定义类型，以及它如何适合现有管道。

# toy 各个example的概述

## toy 1的概述

略

## toy 2(发射基本 MLIR)的概述

发射基本 MLIR，Emitting Basic MLIR

现在我们已经熟悉了我们的语言和 AST，让我们看看 MLIR 如何帮助编译 Toy。

### 简介：多级中间表示

其他编译器，如LLVM（参见Kaleidoscope教程），提供了一组固定的预定义类型和（通常是低层级/类似RISC）指令。在发射 LLVM IR 之前，由给定语言的前端执行任何特定于语言的类型检查、分析或转换。例如，Clang 将使用其 AST 不仅执行静态分析，还执行转换，例如通过 AST 克隆和重写来实现C++模板实例化。最后，构造级别高于 C/C++ 的语言可能需要从其 AST 中非平凡的降低来生成 LLVM IR。

因此，多个前端最终会重新实现基础设施的重要部分，以支持这些分析和转换的需求。MLIR 通过针对可扩展性进行设计来解决此问题。因此，很少有预定义的指令（MLIR术语中的操作）或类型。

### 与 MLIR 接口

MLIR 旨在成为完全可扩展的基础设施;没有一组封闭的属性（想想：常量元数据）、操作或类型。MLIR 通过方言的概念支持这种可扩展性。方言为唯一namespace下的抽象提供了一种分组机制。

在MLIR中，操作是抽象和计算的核心单元，在许多方面类似于LLVM指令。操作可以具有特定于应用程序的语义，并可用于表示LLVM中的所有核心IR结构：指令，全局变量（如函数），模块等。

以下是toy transpose操作的 MLIR 程序集：

|  |
| --- |
| %t\_tensor = "toy.transpose"(%tensor) {inplace = true} : (tensor<2x3xf64>) -> tensor<3x2xf64> loc("example/file/path":12:1) |

让我们分解一下这个MLIR操作的结构：

* %t\_tensor

为此操作定义的结果指定的名称（[其中包括为避免冲突而添加前缀的符号](https://mlir.llvm.org/docs/LangRef/" \l "identifiers-and-keywords)）。一个操作可以定义零个或多个结果（在 Toy 的context中，我们将自己限制为单个结果操作），它们是 SSA 值。该名称在解析过程中使用，但不是持久性的（例如，不会在 SSA 值的内存中表示形式中跟踪它）。

* "toy.transpose"

操作的名称。它应该是一个唯一的字符串，方言的命名空间在“.”之前作为前缀。这可以理解为toy方言中的transpose操作。

* (%tensor)

零个或多个输入操作数（或参数）的列表，这些操作数是由其他操作定义的 SSA 值或引用块参数。

* { inplace = true }

包含零个或多个属性的字典，这些属性是始终恒定的特殊操作数。在这里，我们定义一个名为“inplace”的布尔属性，其常量值为 true。

* (tensor<2x3xf64>) -> tensor<3x2xf64>

这是指函数形式的操作类型，小括号中的参数类型以及后面返回值的类型。

* loc("example/file/path":12:1)

这是源代码中发起此操作的位置。

此处显示的是操作的一般形式。如上所述，MLIR 中的操作集是可扩展的。操作使用一小组概念进行建模，使操作能够进行推理和一般操作。这些概念是：

* 操作的名称。
* SSA 操作数值的列表。
* 属性列表。
* 结果值的类型列表。
* 用于调试的源位置。
* 后续区块列表（主要是分支）。
* 区域列表（用于函数等结构操作）。

在 MLIR 中，每个操作都有一个与之关联的强制源位置。与 LLVM 相反，其中调试信息位置是元数据并且可以删除，而在 MLIR 中，位置是核心要求，API 依赖于并对其进行操作。因此，删除位置是一个明确的选择，不会错误地发生。

举例说明：如果转换将操作替换为另一个操作，则该新操作仍必须附加位置。这样就可以跟踪该操作的来源。

值得注意的是，mlir-opt 工具（用于测试编译器传递的工具）默认情况下不包含输出中的位置。-mlir-print-debuginfo 标志指定包含位置。（运行 mlir-opt --help 以获取更多选项）。

#### 不透明的接口

MLIR 旨在允许自定义所有 IR 元素，例如属性、操作和类型。同时，IR 元素总是可以还原为上述基本概念。这允许 MLIR 为任何 操作解析、表示和 往返IR 。例如，我们可以将上面的 Toy 操作放入一个 .mlir 文件中，并通过mlir-opt进行往返，而无需注册任何 toy 相关方言：

|  |
| --- |
| func.func @toy\_func(%tensor: tensor<2x3xf64>) -> tensor<3x2xf64> {  %t\_tensor = "toy.transpose"(%tensor) { inplace = true } : (tensor<2x3xf64>) -> tensor<3x2xf64>  return %t\_tensor : tensor<3x2xf64>  } |

在未注册的属性、操作和类型的情况下，MLIR 将强制执行一些结构约束（例如支配等），但除此之外它们是完全不透明的。例如，MLIR 几乎没有关于未注册的操作是否可以对特定数据类型进行操作、它可以接受多少个操作数或它产生多少结果的信息。这种灵活性对于bootstrapping目的很有用，但通常不建议在成熟系统中使用。必须通过转换和分析保守地对待未注册的操作，并且它们更难构建和操作。

这种处理可以通过为 Toy 制作一个无效的 IR 并在不触发验证器的情况下往返观察它来观察：

|  |
| --- |
| func.func @main() {  %0 = "toy.print"() : () -> tensor<2x3xf64>  } |

这里有多个问题：toy.print操作不是terminator；它应该有一个操作数；它不应该返回任何值。在下一节中，我们将使用 MLIR 注册我们的方言和操作，插入验证器，并添加更好的 API 来操纵我们的操作。

### 定义toy方言

为了有效地与 MLIR 交互，我们将定义一个新的 Toy 方言。这种方言将模拟 Toy 语言的结构，并为高级分析和转换提供一个简单的途径。

|  |
| --- |
| /// This is the definition of the Toy dialect. A dialect inherits from  /// mlir::Dialect and registers custom attributes, operations, and types. It can  /// also override virtual methods to change some general behavior, which will be  /// demonstrated in later chapters of the tutorial.  class ToyDialect : public mlir::Dialect {  public:  explicit ToyDialect(mlir::MLIRContext \*ctx);  /// Provide a utility accessor to the dialect namespace.  static llvm::StringRef getDialectNamespace() { return "toy"; }  /// An initializer called from the constructor of ToyDialect that is used to  /// register attributes, operations, types, and more within the Toy dialect.  void initialize();  }; |

这是方言的 C++ 定义，但 MLIR 还支持通过 tablegen 以声明方式定义方言。使用声明性规范更加简洁，因为它在定义新方言时消除了对大部分样板的需求。它还可以轻松生成方言文档，可以直接与方言一起描述。在这种声明式格式中，toy方言将被指定为：

|  |
| --- |
| // Provide a definition of the 'toy' dialect in the ODS framework so that we  // can define our operations.  def Toy\_Dialect : Dialect {  // The namespace of our dialect, this corresponds 1-1 with the string we  // provided in `ToyDialect::getDialectNamespace`.  let name = "toy";  // A short one-line summary of our dialect.  let summary = "A high-level dialect for analyzing and optimizing the "  "Toy language";  // A much longer description of our dialect.  let description = [{  The Toy language is a tensor-based language that allows you to define  functions, perform some math computation, and print results. This dialect  provides a representation of the language that is amenable to analysis and  optimization.  }];  // The C++ namespace that the dialect class definition resides in.  let cppNamespace = "toy";  } |

(个人理解，这就是对应的方言td文件，通过mlir-tblgen程序来生成)

要查看生成的内容，我们可以mlir-tblgen使用如下操作运行命令 gen-dialect-decls：

|  |
| --- |
| ${build\_root}/bin/mlir-tblgen -gen-dialect-decls ${mlir\_src\_root}/examples/toy/Ch2/include/toy/Ops.td -I ${mlir\_src\_root}/include/ |

定义方言后，现在可以将其加载到 MLIRContext 中：

|  |
| --- |
| context.loadDialect<ToyDialect>(); |

默认情况下， 一个MLIRContext仅加载 Builtin Dialect  ，它提供了一些核心 IR 组件，这意味着必须显式加载其他方言，例如我们的toy方言。

### 定义toy操作

现在我们有了Toy方言，我们可以开始定义操作了。这将允许提供系统其余部分可以挂钩的语义信息。作为示例，让我们来看看toy.constant 操作的创建。此操作将表示 Toy 语言中的常量值。

|  |
| --- |
| %4 = "toy.constant"() {value = dense<1.0> : tensor<2x3xf64>} : () -> tensor<2x3xf64> |

此操作采用零操作数，一个名为value代表常量值的密集元素属性 ，并返回RankedTensorType 的单个结果 。操作类继承自 CRTP  mlir::Op 类，该类还采用一些可选 Traits 来自定义其行为。Traits 是一种机制，我们可以通过它向 Operation 中注入额外的行为，例如额外的访问器、验证等。让我们在下面看一下我们上面描述的常量操作的可能定义：

|  |
| --- |
| class ConstantOp : public mlir::Op<  /// `mlir::Op` is a CRTP class, meaning that we provide the  /// derived class as a template parameter.  ConstantOp,  /// The ConstantOp takes zero input operands.  mlir::OpTrait::ZeroOperands,  /// The ConstantOp returns a single result.  mlir::OpTrait::OneResult,  /// We also provide a utility `getType` accessor that  /// returns the TensorType of the single result.  mlir::OpTraits::OneTypedResult<TensorType>::Impl> {  public:  /// Inherit the constructors from the base Op class.  using Op::Op;  /// Provide the unique name for this operation. MLIR will use this to register  /// the operation and uniquely identify it throughout the system. The name  /// provided here must be prefixed by the parent dialect namespace followed  /// by a `.`.  static llvm::StringRef getOperationName() { return "toy.constant"; }  /// Return the value of the constant by fetching it from the attribute.  mlir::DenseElementsAttr getValue();  /// Operations may provide additional verification beyond what the attached  /// traits provide. Here we will ensure that the specific invariants of the  /// constant operation are upheld, for example the result type must be  /// of TensorType and matches the type of the constant `value`.  LogicalResult verifyInvariants();  /// Provide an interface to build this operation from a set of input values.  /// This interface is used by the `builder` classes to allow for easily  /// generating instances of this operation:  /// mlir::OpBuilder::create<ConstantOp>(...)  /// This method populates the given `state` that MLIR uses to create  /// operations. This state is a collection of all of the discrete elements  /// that an operation may contain.  /// Build a constant with the given return type and `value` attribute.  static void build(mlir::OpBuilder &builder, mlir::OperationState &state,  mlir::Type result, mlir::DenseElementsAttr value);  /// Build a constant and reuse the type from the given 'value'.  static void build(mlir::OpBuilder &builder, mlir::OperationState &state,  mlir::DenseElementsAttr value);  /// Build a constant by broadcasting the given 'value'.  static void build(mlir::OpBuilder &builder, mlir::OperationState &state,  double value);  }; |

我们可以在ToyDialect初始化器中注册此操作：

|  |
| --- |
| void ToyDialect::initialize() {  addOperations<ConstantOp>();  } |

#### Op vs Operation：使用MLIR操作

现在我们已经定义了一个操作，我们将要访问和转换它。在 MLIR 中，有两个与操作相关的主要类：Operation和Op。该类Operation用于对所有操作进行通用建模。它是“不透明的”，因为它不描述特定操作或操作类型的属性。相反，Operation该类将通用 API 提供给操作实例。另一方面，每个特定类型的操作都由一个Op派生类表示。例如ConstantOp 表示具有零个输入和一个输出的操作，它始终设置为相同的值。Op派生类充当了一个 Operation\*的智能指针封装，提供特定于操作的访问器方法，以及操作的类型安全属性。这意味着当我们定义我们的 Toy 操作时，我们只是在定义一个干净的、语义上有用的接口来构建类，并与Operation类交互。这就是我们ConstantOp class不定义类字段的原因；此操作的所有数据都存储在引用的 Operation 中. 这种设计的一个副作用是我们总是 “by-value”传递Op派生类，而不是通过引用或指针（按值传递是 MLIR 中的常见习惯用法，并且类似地适用于属性、类型等）。给定一个通用Operation\*实例，我们总是可 使用 LLVM 的转换基础设施获得一个特定Op实例：

|  |
| --- |
| void processConstantOp(mlir::Operation \*operation) {  ConstantOp op = llvm::dyn\_cast<ConstantOp>(operation);  // This operation is not an instance of `ConstantOp`.  if (!op)  return;  // Get the internal operation instance wrapped by the smart pointer.  mlir::Operation \*internalOperation = op.getOperation();  assert(internalOperation == operation &&  "these operation instances are the same");  } |

#### 使用操作定义规范（ODS）框架

除了特化mlir::Op C++模板之外，mlir还支持以声明方式定义操作。这是通过 操作定义规范框架实现的。mlir::Op有关操作的事实被简洁地指定到 TableGen 记录中，该记录将在编译时扩展为等效的C++ 模板特化。使用 ODS 框架是在 MLIR 中定义操作的理想方式，因为它在面向 C++ API 更改时具有简单、简洁和总体稳定性。

让我们看看如何定义与我们的 ConstantOp 等效的 ODS：

ODS 中的Op操作是通过从Op类继承来定义的。为了简化我们的操作定义，我们将为玩具方言中的操作定义一个基类。

|  |
| --- |
| // Base class for toy dialect operations. This operation inherits from the base  // `Op` class in OpBase.td, and provides:  // \* The parent dialect of the operation.  // \* The mnemonic for the operation, or the name without the dialect prefix.  // \* A list of traits for the operation.  class Toy\_Op<string mnemonic, list<Trait> traits = []> :  Op<Toy\_Dialect, mnemonic, traits>; |

定义了所有初步部分后，我们就可以开始定义常量操作了。

我们通过继承上面的基类“Toy\_Op”来定义玩具操作。在这里，我们提供操作的助记符和特征列表。这里的 助记词与 ConstantOp::getOperationName不带方言前缀的助记词相匹配；toy.. 我们的 C++ 定义中缺少了 theZeroOperands和OneResult traits；这些将根据我们稍后定义的arguments和results字段自动推断。

|  |
| --- |
| def ConstantOp : Toy\_Op<"constant"> {  } |

此时您可能想知道 TableGen 生成的 C++ 代码是什么样子的。只需运行带有gen-op-decls或gen-op-defs的mlir-tblgen命令， 如下所示：

|  |
| --- |
| ${build\_root}/bin/mlir-tblgen -gen-op-defs  ${mlir\_src\_root}/examples/toy/Ch2/include/toy/Ops.td -I ${mlir\_src\_root}/include/ |

根据所选的操作，这将打印ConstantOp类声明或其实现。在开始使用 TableGen 时，将此输出与手工实现进行比较非常有用。

##### 定义参数和结果

定义操作的外壳后，我们现在可以为我们的操作提供 输入和 输出。操作的输入或参数可以是 SSA 操作数值的属性或类型。结果对应于操作产生的值的一组类型：

|  |
| --- |
| def ConstantOp : Toy\_Op<"constant"> {  // The constant operation takes an attribute as the only input.  // `F64ElementsAttr` corresponds to a 64-bit floating-point ElementsAttr.  let arguments = (ins F64ElementsAttr:$value);  // The constant operation returns a single value of TensorType.  // F64Tensor corresponds to a 64-bit floating-point TensorType.  let results = (outs F64Tensor);  } |

通过为参数或结果提供名称，例如$value，ODS 将自动生成一个匹配的访问器：DenseElementsAttr ConstantOp::value()。

##### 添加文档

定义操作后的下一步是记录它。操作可以提供 summary和description 字段来描述操作的语义。此信息对方言用户很有用，甚至可用于自动生成 Markdown 文档。

|  |
| --- |
| def ConstantOp : Toy\_Op<"constant"> {  // Provide a summary and description for this operation. This can be used to  // auto-generate documentation of the operations within our dialect.  let summary = "constant operation";  let description = [{  Constant operation turns a literal into an SSA value. The data is attached  to the operation as an attribute. For example:  %0 = "toy.constant"()  { value = dense<[[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]]> : tensor<2x3xf64> }  : () -> tensor<2x3xf64>  }];  // The constant operation takes an attribute as the only input.  // `F64ElementsAttr` corresponds to a 64-bit floating-point ElementsAttr.  let arguments = (ins F64ElementsAttr:$value);  // The generic call operation returns a single value of TensorType.  // F64Tensor corresponds to a 64-bit floating-point TensorType.  let results = (outs F64Tensor);  } |

##### 验证操作语义

至此，我们已经涵盖了大部分原始 C++ 操作定义。下一个要定义的部分是验证器。幸运的是，与命名访问器非常相似，ODS 框架将根据我们提供的约束自动生成大量必要的验证逻辑。这意味着我们不需要验证返回类型的结构，甚至不需要验证输入属性value。在许多情况下，ODS 操作甚至不需要额外的验证。要添加额外的验证逻辑，操作可以覆盖该 verifier 字段。该verifier字段允许定义一个 C++ code blob， 该code blob将作为ConstantOp::verify 的一部分来运行.。这个 blob 可以假定操作的所有其他不变量都已经过验证：

|  |
| --- |
| def ConstantOp : Toy\_Op<"constant"> {  // Provide a summary and description for this operation. This can be used to  // auto-generate documentation of the operations within our dialect.  let summary = "constant operation";  let description = [{  Constant operation turns a literal into an SSA value. The data is attached  to the operation as an attribute. For example:  %0 = "toy.constant"()  { value = dense<[[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]]> : tensor<2x3xf64> }  : () -> tensor<2x3xf64>  }];  // The constant operation takes an attribute as the only input.  // `F64ElementsAttr` corresponds to a 64-bit floating-point ElementsAttr.  let arguments = (ins F64ElementsAttr:$value);  // The generic call operation returns a single value of TensorType.  // F64Tensor corresponds to a 64-bit floating-point TensorType.  let results = (outs F64Tensor);  // Add additional verification logic to the constant operation. Setting this bit  // to `1` will generate a `::mlir::LogicalResult verify()` declaration on the  // operation class that is called after ODS constructs have been verified, for  // example the types of arguments and results. We implement additional verification  // in the definition of this `verify` method in the C++ source file.  let hasVerifier = 1;  } |

##### 附加build方法

我们最初的 C++ 示例中最后缺少的组件是build 方法。ODS可以自动生成一些简单的构建方法，这里会为我们生成第一个构建方法。对于其余部分，我们定义 builders字段。该字段采用OpBuilder对象列表，这些对象采用与 C++ 参数列表相对应的字符串，以及可用于指定内联实现的可选代码块。

|  |
| --- |
| def ConstantOp : Toy\_Op<"constant"> {  ...  // Add custom build methods for the constant operation. These methods populate  // the `state` that MLIR uses to create operations, i.e. these are used when  // using `builder.create<ConstantOp>(...)`.  let builders = [  // Build a constant with a given constant tensor value.  OpBuilder<(ins "DenseElementsAttr":$value), [{  // Call into an autogenerated `build` method.  build(builder, result, value.getType(), value);  }]>,  // Build a constant with a given constant floating-point value. This builder  // creates a declaration for `ConstantOp::build` with the given parameters.  OpBuilder<(ins "double":$value)>  ];  } |

##### 指定自定义程序集格式

此时我们可以生成我们的“Toy IR”。例如，以下内容：

|  |
| --- |
| # User defined generic function that operates on unknown shaped arguments.  def multiply\_transpose(a, b) {  return transpose(a) \* transpose(b);  }  def main() {  var a<2, 3> = [[1, 2, 3], [4, 5, 6]];  var b<2, 3> = [1, 2, 3, 4, 5, 6];  var c = multiply\_transpose(a, b);  var d = multiply\_transpose(b, a);  print(d);  } |

结果如下 IR：

|  |
| --- |
| module {  "toy.func"() ({  ^bb0(%arg0: tensor<\*xf64> loc("test/Examples/Toy/Ch2/codegen.toy":4:1), %arg1: tensor<\*xf64> loc("test/Examples/Toy/Ch2/codegen.toy":4:1)):  %0 = "toy.transpose"(%arg0) : (tensor<\*xf64>) -> tensor<\*xf64> loc("test/Examples/Toy/Ch2/codegen.toy":5:10)  %1 = "toy.transpose"(%arg1) : (tensor<\*xf64>) -> tensor<\*xf64> loc("test/Examples/Toy/Ch2/codegen.toy":5:25)  %2 = "toy.mul"(%0, %1) : (tensor<\*xf64>, tensor<\*xf64>) -> tensor<\*xf64> loc("test/Examples/Toy/Ch2/codegen.toy":5:25)  "toy.return"(%2) : (tensor<\*xf64>) -> () loc("test/Examples/Toy/Ch2/codegen.toy":5:3)  }) {sym\_name = "multiply\_transpose", type = (tensor<\*xf64>, tensor<\*xf64>) -> tensor<\*xf64>} : () -> () loc("test/Examples/Toy/Ch2/codegen.toy":4:1)  "toy.func"() ({  %0 = "toy.constant"() {value = dense<[[1.000000e+00, 2.000000e+00, 3.000000e+00], [4.000000e+00, 5.000000e+00, 6.000000e+00]]> : tensor<2x3xf64>} : () -> tensor<2x3xf64> loc("test/Examples/Toy/Ch2/codegen.toy":9:17)  %1 = "toy.reshape"(%0) : (tensor<2x3xf64>) -> tensor<2x3xf64> loc("test/Examples/Toy/Ch2/codegen.toy":9:3)  %2 = "toy.constant"() {value = dense<[1.000000e+00, 2.000000e+00, 3.000000e+00, 4.000000e+00, 5.000000e+00, 6.000000e+00]> : tensor<6xf64>} : () -> tensor<6xf64> loc("test/Examples/Toy/Ch2/codegen.toy":10:17)  %3 = "toy.reshape"(%2) : (tensor<6xf64>) -> tensor<2x3xf64> loc("test/Examples/Toy/Ch2/codegen.toy":10:3)  %4 = "toy.generic\_call"(%1, %3) {callee = @multiply\_transpose} : (tensor<2x3xf64>, tensor<2x3xf64>) -> tensor<\*xf64> loc("test/Examples/Toy/Ch2/codegen.toy":11:11)  %5 = "toy.generic\_call"(%3, %1) {callee = @multiply\_transpose} : (tensor<2x3xf64>, tensor<2x3xf64>) -> tensor<\*xf64> loc("test/Examples/Toy/Ch2/codegen.toy":12:11)  "toy.print"(%5) : (tensor<\*xf64>) -> () loc("test/Examples/Toy/Ch2/codegen.toy":13:3)  "toy.return"() : () -> () loc("test/Examples/Toy/Ch2/codegen.toy":8:1)  }) {sym\_name = "main", type = () -> ()} : () -> () loc("test/Examples/Toy/Ch2/codegen.toy":8:1)  } loc(unknown) |

这里要注意的一件事是我们所有的玩具操作都是使用通用程序集格式打印的。这种格式就是本章开头分解toy.transpose时显示的格式 。MLIR 允许操作通过 C++以声明方式或命令方式定义自己的自定义程序集格式 。定义自定义程序集格式，允许通过删除通用格式所需的大量小东西(fluff)，来裁剪通用的 IR 到更具可读性的定制IR。让我们来看一个我们想要简化的操作格式的例子。

**toy.print**

当前的形式toy.print有点冗长。我们想去掉很多额外的字符。让我们首先考虑什么是toy.print的好的格式，然后看看我们如何实现它。看看toy.print我们得到的基础知识：

|  |
| --- |
| toy.print %5 : tensor<\*xf64> loc(...) |

在这里，我们已经将大部分格式剥离到最基本的部分，并且它变得更具可读性。要提供自定义程序集格式，操作可以为C++ 格式覆盖hasCustomAssemblyFormat 字段， 或者为声明格式覆盖 assemblyFormat字段。让我们首先看一下 C++ 变体，因为这是声明格式在内部映射的内容。

|  |
| --- |
| /// Consider a stripped definition of `toy.print` here.  def PrintOp : Toy\_Op<"print"> {  let arguments = (ins F64Tensor:$input);  // Divert the printer and parser to `parse` and `print` methods on our operation,  // to be implemented in the .cpp file. More details on these methods is shown below.  let hasCustomAssemblyFormat = 1;  } |

printer 和parser的 C++ 实现如下所示：

|  |
| --- |
| /// The 'OpAsmPrinter' class is a stream that will allows for formatting  /// strings, attributes, operands, types, etc.  void PrintOp::print(mlir::OpAsmPrinter &printer) {  printer << "toy.print " << op.input();  printer.printOptionalAttrDict(op.getAttrs());  printer << " : " << op.input().getType();  }  /// The 'OpAsmParser' class provides a collection of methods for parsing  /// various punctuation, as well as attributes, operands, types, etc. Each of  /// these methods returns a `ParseResult`. This class is a wrapper around  /// `LogicalResult` that can be converted to a boolean `true` value on failure,  /// or `false` on success. This allows for easily chaining together a set of  /// parser rules. These rules are used to populate an `mlir::OperationState`  /// similarly to the `build` methods described above.  mlir::ParseResult PrintOp::parse(mlir::OpAsmParser &parser,  mlir::OperationState &result) {  // Parse the input operand, the attribute dictionary, and the type of the  // input.  mlir::OpAsmParser::UnresolvedOperand inputOperand;  mlir::Type inputType;  if (parser.parseOperand(inputOperand) ||  parser.parseOptionalAttrDict(result.attributes) || parser.parseColon() ||  parser.parseType(inputType))  return mlir::failure();  // Resolve the input operand to the type we parsed in.  if (parser.resolveOperand(inputOperand, inputType, result.operands))  return mlir::failure();  return mlir::success();  } |

定义了 C++ 实现后，让我们看看如何将其映射到 声明格式。声明格式主要由三个不同的部分组成：

* 指令Directives
* 一种内置函数，带有一组可选的参数。
* 字面量Literals
* 用``包起来的关键字或标点符号。
* 变量
* 已在操作本身上注册的实体，即参数（属性或操作数）、结果、successor等。在上面的PrintOp 示例中，变量是$input。

C++ 格式的直接映射类似于：

|  |
| --- |
| /// Consider a stripped definition of `toy.print` here.  def PrintOp : Toy\_Op<"print"> {  let arguments = (ins F64Tensor:$input);  // In the following format we have two directives, `attr-dict` and `type`.  // These correspond to the attribute dictionary and the type of a given  // variable represectively.  let assemblyFormat = "$input attr-dict `:` type($input)";  } |

声明式格式有许多更有趣的 特性，所以在用 C++ 实现自定义格式之前一定要检查一下。在美化了一些操作的格式之后，我们现在的可读性更高了：

|  |
| --- |
| module {  toy.func @multiply\_transpose(%arg0: tensor<\*xf64>, %arg1: tensor<\*xf64>) -> tensor<\*xf64> {  %0 = toy.transpose(%arg0 : tensor<\*xf64>) to tensor<\*xf64> loc("test/Examples/Toy/Ch2/codegen.toy":5:10)  %1 = toy.transpose(%arg1 : tensor<\*xf64>) to tensor<\*xf64> loc("test/Examples/Toy/Ch2/codegen.toy":5:25)  %2 = toy.mul %0, %1 : tensor<\*xf64> loc("test/Examples/Toy/Ch2/codegen.toy":5:25)  toy.return %2 : tensor<\*xf64> loc("test/Examples/Toy/Ch2/codegen.toy":5:3)  } loc("test/Examples/Toy/Ch2/codegen.toy":4:1)  toy.func @main() {  %0 = toy.constant dense<[[1.000000e+00, 2.000000e+00, 3.000000e+00], [4.000000e+00, 5.000000e+00, 6.000000e+00]]> : tensor<2x3xf64> loc("test/Examples/Toy/Ch2/codegen.toy":9:17)  %1 = toy.reshape(%0 : tensor<2x3xf64>) to tensor<2x3xf64> loc("test/Examples/Toy/Ch2/codegen.toy":9:3)  %2 = toy.constant dense<[1.000000e+00, 2.000000e+00, 3.000000e+00, 4.000000e+00, 5.000000e+00, 6.000000e+00]> : tensor<6xf64> loc("test/Examples/Toy/Ch2/codegen.toy":10:17)  %3 = toy.reshape(%2 : tensor<6xf64>) to tensor<2x3xf64> loc("test/Examples/Toy/Ch2/codegen.toy":10:3)  %4 = toy.generic\_call @multiply\_transpose(%1, %3) : (tensor<2x3xf64>, tensor<2x3xf64>) -> tensor<\*xf64> loc("test/Examples/Toy/Ch2/codegen.toy":11:11)  %5 = toy.generic\_call @multiply\_transpose(%3, %1) : (tensor<2x3xf64>, tensor<2x3xf64>) -> tensor<\*xf64> loc("test/Examples/Toy/Ch2/codegen.toy":12:11)  toy.print %5 : tensor<\*xf64> loc("test/Examples/Toy/Ch2/codegen.toy":13:3)  toy.return loc("test/Examples/Toy/Ch2/codegen.toy":8:1)  } loc("test/Examples/Toy/Ch2/codegen.toy":8:1)  } loc(unknown) |

上面我们介绍了在 ODS 框架中定义操作的几个概念，但还有更多我们没有机会了解的概念：区域、可变操作数等。查看 完整规范了解更多详细信息。

### 完整的玩具示例

我们现在可以生成我们的“Toy IR”。您可以在上面的示例中构建toyc-ch2并尝试自己toyc-ch2 test/Examples/Toy/Ch2/codegen.toy -emit=mlir -mlir-print-debuginfo。我们还可以检查我们的 RoundTrip: toyc-ch2 test/Examples/Toy/Ch2/codegen.toy -emit=mlir -mlir-print-debuginfo 2> codegen.mlir 后跟 toyc-ch2 codegen.mlir -emit=mlir。 您还应该使用mlir-tblgen最终定义文件并研究生成的 C++ 代码。

此时，MLIR 知道了我们的 Toy 方言和操作。在 下一章中，我们将利用我们的新方言为 Toy 语言实施一些特定于高级语言的分析和转换。

## 特定于高级语言的分析和转换

创建一种能够近似表示输入语言语义的方言，该方言可以在 MLIR 中进行分析、转换和优化，这些分析、转换和优化需要高级语言信息，并且通常在AST 语言 上执行。例如，在 C++ 中clang有一个相当 重的机制， 该机制用于 执行模板实例化。

我们将编译器转换分为两类： 局部和全局。在本章中，我们将重点关注如何利用 Toy Dialect 及其高级语义来执行在 LLVM 中难以实现的局部模式匹配转换。为此，我们使用 MLIR 的  [Generic DAG Rewriter](https://mlir.llvm.org/docs/PatternRewriter/)。

有两种方法可用于实现模式匹配转换： 1. 命令式、C++ 模式匹配和重写 2. 声明式、基于规则的模式匹配和重写，使用表驱动的[声明性重写规则](https://mlir.llvm.org/docs/DeclarativeRewrites/)(DRR，Declarative Rewrite Rules )。请注意，使用 DRR 需要使用 ODS 定义操作，如 第 2 章所述。

### 使用 C++ 样式模式匹配和重写来优化 Transpose

让我们从一个简单的模式开始，并尝试消除抵消的两个转置序列：transpose(transpose(X)) -> X。这是相应的玩具示例：

|  |
| --- |
| def transpose\_transpose(x) {  return transpose(transpose(x));  } |

对应于以下IR：

|  |
| --- |
| toy.func @transpose\_transpose(%arg0: tensor<\*xf64>) -> tensor<\*xf64> {  %0 = toy.transpose(%arg0 : tensor<\*xf64>) to tensor<\*xf64>  %1 = toy.transpose(%0 : tensor<\*xf64>) to tensor<\*xf64>  toy.return %1 : tensor<\*xf64>  } |

这是一个很好的转换示例，在 Toy IR 上匹配起来很简单，但对于 LLVM 来说却很难计算。例如，今天 Clang 无法优化掉临时数组，并且使用简单转置的计算用这些循环表示：

|  |
| --- |
| #define N 100  #define M 100  void sink(void \*);  void double\_transpose(int A[N][M]) {  int B[M][N];  for(int i = 0; i < N; ++i) {  for(int j = 0; j < M; ++j) {  B[j][i] = A[i][j];  }  }  for(int i = 0; i < N; ++i) {  for(int j = 0; j < M; ++j) {  A[i][j] = B[j][i];  }  }  sink(A);  } |

对于一种简单的 C++ 重写方法，涉及在 IR 中匹配树状模式并将其替换为一组不同的操作，我们可以通过实现一个RewritePattern 插入 到 MLIR Canonicalizer pass：

|  |
| --- |
| /// Fold transpose(transpose(x)) -> x  struct SimplifyRedundantTranspose : public mlir::OpRewritePattern<TransposeOp> {  /// We register this pattern to match every toy.transpose in the IR.  /// The "benefit" is used by the framework to order the patterns and process  /// them in order of profitability.  SimplifyRedundantTranspose(mlir::MLIRContext \*context)  : OpRewritePattern<TransposeOp>(context, /\*benefit=\*/1) {}  /// This method is attempting to match a pattern and rewrite it. The rewriter  /// argument is the orchestrator of the sequence of rewrites. It is expected  /// to interact with it to perform any changes to the IR from here.  mlir::LogicalResult  matchAndRewrite(TransposeOp op,  mlir::PatternRewriter &rewriter) const override {  // Look through the input of the current transpose.  mlir::Value transposeInput = op.getOperand();  TransposeOp transposeInputOp = transposeInput.getDefiningOp<TransposeOp>();  // Input defined by another transpose? If not, no match.  if (!transposeInputOp)  return failure();  // Otherwise, we have a redundant transpose. Use the rewriter.  rewriter.replaceOp(op, {transposeInputOp.getOperand()});  return success();  }  }; |

这个rewriter的实现是在ToyCombine.cpp.  [canonicalization pass](https://mlir.llvm.org/docs/Canonicalization/) 以贪婪、迭代的方式应用由操作定义的转换。 为确保规范化过程应用我们的新转换，我们设置[hasCanonicalizer = 1](https://mlir.llvm.org/docs/DefiningDialects/Operations/" \l "hascanonicalizer)并将模式注册到规范化框架。

|  |
| --- |
| // Register our patterns for rewrite by the Canonicalization framework.  void TransposeOp::getCanonicalizationPatterns(  RewritePatternSet &results, MLIRContext \*context) {  results.add<SimplifyRedundantTranspose>(context);  } |

我们还需要更新我们的主文件toyc.cpp，以添加优化管道。在 MLIR 中，优化通过PassManager与 LLVM 类似的方式运行：

|  |
| --- |
| mlir::PassManager pm(module.getContext());  pm.addNestedPass<mlir::toy::FuncOp>(mlir::createCanonicalizerPass()); |

最后，我们可以运行toyc-ch3 test/Examples/Toy/Ch3/transpose\_transpose.toy -emit=mlir -opt并观察我们的模式

|  |
| --- |
| toy.func @transpose\_transpose(%arg0: tensor<\*xf64>) -> tensor<\*xf64> {  %0 = toy.transpose(%arg0 : tensor<\*xf64>) to tensor<\*xf64>  toy.return %arg0 : tensor<\*xf64>  } |

正如预期的那样，我们现在直接返回函数参数，绕过任何转置操作。然而，其中一个转置仍然没有被消除。那不理想！发生的事情是，我们的模式用函数输入替换了最后一个变换，并留下了现在死掉的转置输入。Canonicalizer 知道清理死操作；然而，MLIR 保守地假设操作可能有副作用。我们可以通过向我们的 TransposeOp 中添加一个新特征 NoMemoryEffect 来解决这个问题：

|  |
| --- |
| def TransposeOp : Toy\_Op<"transpose", [NoMemoryEffect]> {...} |

让我们现在重试toyc-ch3 test/transpose\_transpose.toy -emit=mlir -opt：

|  |
| --- |
| toy.func @transpose\_transpose(%arg0: tensor<\*xf64>) -> tensor<\*xf64> {  toy.return %arg0 : tensor<\*xf64>  } |

完美的！没有transpose留下任何操作 - 代码是最佳的。

在下一节中，我们将 DRR 用于与 Reshape 操作相关的模式匹配优化。

### 使用 DRR 优化reshapes

声明式、基于规则的模式匹配和重写 (DRR) 是一种基于 DAG 的声明式rewriter操作，它为模式匹配和重写规则提供基于表的语法：

|  |
| --- |
| class Pattern<  dag sourcePattern, list<dag> resultPatterns,  list<dag> additionalConstraints = [],  dag benefitsAdded = (addBenefit 0)>; |

类似于 SimplifyRedundantTranspose 的冗余reshape优化可以使用 DRR 更简单地表示如下：

|  |
| --- |
| // Reshape(Reshape(x)) = Reshape(x)  def ReshapeReshapeOptPattern : Pat<(ReshapeOp(ReshapeOp $arg)),  (ReshapeOp $arg)>; |

可以在 path/to/BUILD/tools/mlir/examples/toy/Ch3/ToyCombine.inc 下找到与每个 DRR 模式对应的自动生成的 C++ 代码。

当转换以参数和结果的某些属性为条件时，DRR 还提供了一种添加参数约束的方法。一个例子是，在冗余时，即当输入和输出形状相同时，消除reshape的转换。

|  |
| --- |
| def TypesAreIdentical : Constraint<CPred<"$0.getType() == $1.getType()">>;  def RedundantReshapeOptPattern : Pat<  (ReshapeOp:$res $arg), (replaceWithValue $arg),  [(TypesAreIdentical $res, $arg)]>; |

一些优化可能需要对指令参数进行额外的转换。这是使用 NativeCodeCall 实现的，它允许通过调用 C++ 辅助函数或使用内联 C++ 进行更复杂的转换。这种优化的一个例子是 FoldConstantReshape，我们通过就地reshape常量并消除reshape操作来优化常量值的 Reshape。

|  |
| --- |
| def ReshapeConstant : NativeCodeCall<"$0.reshape(($1.getType()).cast<ShapedType>())">;  def FoldConstantReshapeOptPattern : Pat<  (ReshapeOp:$res (ConstantOp $arg)),  (ConstantOp (ReshapeConstant $arg, $res))>; |

我们使用以下 trivial\_reshape.toy 程序演示这些reshape优化：

|  |
| --- |
| def main() {  var a<2,1> = [1, 2];  var b<2,1> = a;  var c<2,1> = b;  print(c);  }  module {  toy.func @main() {  %0 = toy.constant dense<[1.000000e+00, 2.000000e+00]> : tensor<2xf64>  %1 = toy.reshape(%0 : tensor<2xf64>) to tensor<2x1xf64>  %2 = toy.reshape(%1 : tensor<2x1xf64>) to tensor<2x1xf64>  %3 = toy.reshape(%2 : tensor<2x1xf64>) to tensor<2x1xf64>  toy.print %3 : tensor<2x1xf64>  toy.return  }  } |

我们可以尝试运行toyc-ch3 test/Examples/Toy/Ch3/trivial\_reshape.toy -emit=mlir -opt并观察我们的模式：

|  |
| --- |
| module {  toy.func @main() {  %0 = toy.constant dense<[[1.000000e+00], [2.000000e+00]]> : tensor<2x1xf64>  toy.print %0 : tensor<2x1xf64>  toy.return  }  } |

正如预期的那样，规范化后没有 reshape 操作。

有关声明性重写方法的更多详细信息，请参阅 [表驱动的声明性重写规则 (DRR)](https://mlir.llvm.org/docs/DeclarativeRewrites/)。

在本章中，我们看到了如何通过始终可用的钩子来使用某些核心转换。在 下一章中，我们将看到如何使用通过接口更好地扩展的通用解决方案。

## 使用接口启用通用转换

### 背景：应对可扩展的 IR

通过方言， MLIR 允许表示许多不同的抽象层次；我们之前定义的 Toy 方言就是这样一个例子。尽管这些不同的方言可能代表不同的抽象，但通常有一组我们想要执行的通用转换和分析。出现的问题是，天真地为每种方言实现每个转换会导致大量代码重复，因为内部算法通常非常相似，如果不相同的话。我们希望提供转换功能，以不透明的方式挂钩到像 Toy 这样的方言中，以获取他们需要的信息。

MLIR 为某些核心转换提供了一组始终可用的挂钩，如 前一章所示，我们通过操作的挂钩 ( getCanonicalizationPatterns) 注册了一些规范化。然而，这些类型的钩子并不能很好地扩展。因此，设计了一个更通用的解决方案，以 接口的形式，使 MLIR 基础设施像表示一样可扩展。接口为方言和操作提供了一种通用机制，以便为转换或分析提供信息。

### Shape Inference：为代码生成做准备

我们的 Toy IR 目前在通用张量上运行，这意味着除了常量初始化期间我们不知道张量的形状。这使优化和代码生成变得复杂。幸运的是，我们可以通过计算简单地传播形状，直到它们都已知为止。问题是如何处理对用户定义的通用函数的调用：每个调用点都可以推断出不同的形状。一种可能性是根据参数类型执行符号推理，但如果我们要在语言中引入更多控制流，这将很难概括。另一种方法是函数特化，其中每个具有新参数形状的调用点都复制被调用函数并对其进行特化。我们对 Toy 采取的方法是内联所有函数调用，

#### 内联

在这里，我们可以编写专门为 Toy 方言设计的内联算法，但这可能会变得相当复杂，具体取决于我们想要的复杂程度。撇开建模成本不谈，从头开始实施纯结构转换已经很复杂了。值得庆幸的是，MLIR 提供了方言可以插入的通用内联算法。在 Toy 中，我们需要做的就是为内联器提供 接口来挂钩(hook)。

我们需要做的第一件事是在 Toy 方言中定义对内联操作的约束。此信息通过 [方言接口](https://mlir.llvm.org/docs/Interfaces/" \l "dialect-interfaces)提供。这本质上是一个包含一组虚拟挂钩的类，方言可以重载这些虚拟挂钩。在这种情况下，接口是DialectInlinerInterface.

|  |
| --- |
| /// This class defines the interface for handling inlining with Toy operations.  /// We simplify inherit from the base interface class and override  /// the necessary methods.  struct ToyInlinerInterface : public DialectInlinerInterface {  using DialectInlinerInterface::DialectInlinerInterface;  /// This hook checks to see if the given callable operation is legal to inline  /// into the given call. For Toy this hook can simply return true, as the Toy  /// Call operation is always inlinable.  bool isLegalToInline(Operation \*call, Operation \*callable,  bool wouldBeCloned) const final {  return true;  }  /// This hook checks to see if the given operation is legal to inline into the  /// given region. For Toy this hook can simply return true, as all Toy  /// operations are inlinable.  bool isLegalToInline(Operation \*, Region \*, bool,  BlockAndValueMapping &) const final {  return true;  }  /// This hook cheks if the given 'src' region can be inlined into the 'dest'  /// region. The regions here are the bodies of the callable functions. For  /// Toy, any function can be inlined, so we simply return true.  bool isLegalToInline(Region \*dest, Region \*src, bool wouldBeCloned,  BlockAndValueMapping &valueMapping) const final {  return true;  }  /// This hook is called when a terminator operation has been inlined. The only  /// terminator that we have in the Toy dialect is the return  /// operation(toy.return). We handle the return by replacing the values  /// previously returned by the call operation with the operands of the  /// return.  void handleTerminator(Operation \*op,  ArrayRef<Value> valuesToRepl) const final {  // Only "toy.return" needs to be handled here.  auto returnOp = cast<ReturnOp>(op);  // Replace the values directly with the return operands.  assert(returnOp.getNumOperands() == valuesToRepl.size());  for (const auto &it : llvm::enumerate(returnOp.getOperands()))  valuesToRepl[it.index()].replaceAllUsesWith(it.value());  }  }; |

此外，内联器只会丢弃私有可见的未使用函数定义。我们还必须在 MLIR 生成器(MLIR generator)中设置函数（主函数除外）的可见性。

|  |
| --- |
| /// Emit a new function and add it to the MLIR module.  mlir::toy::FuncOp mlirGen(FunctionAST &funcAST) {  ...  // If this function isn't main, then set the visibility to private.  if (funcAST.getProto()->getName() != "main")  function.setPrivate();  return function;  } |

然后我们直接在 Toy 方言上注册我们的方言接口，类似于我们为操作所做的。

|  |
| --- |
| void ToyDialect::initialize() {  addInterfaces<ToyInlinerInterface>();  } |

接下来，我们需要提供一种方法让内联器知道 toy.generic\_call 代表 一个调用，toy.func代表一个函数。MLIR 提供 操作接口，可用于将操作标记为“call-like”或“callable-like”。与 dialect interfaces不同， operation interfaces提供更精细的信息粒度，这些信息是对单个操作而言是特定的和核心的。我们将在此处添加的接口是 CallOpInterface和CallableOpInterface。

要添加此接口，我们只需将定义包含到我们的操作规范文件 ( Ops.td) 中：

|  |
| --- |
| include "mlir/Interfaces/CallInterfaces.td" |

并将其添加到GenericCallOp的 traits 列表中：

|  |
| --- |
| def FuncOp : Toy\_Op<"func",  [DeclareOpInterfaceMethods<CallableOpInterface>]> {  ...  }  def GenericCallOp : Toy\_Op<"generic\_call",  [DeclareOpInterfaceMethods<CallOpInterface>]> {  ...  } |

在上面我们还使用DeclareOpInterfaceMethods指令自动声明了 GenericCallOp 的类声明中的所有接口方法。这意味着我们只需要提供一个定义：

|  |
| --- |
| /// Returns the region on the function operation that is callable.  Region \*FuncOp::getCallableRegion() { return &getBody(); }  /// Returns the results types that the callable region produces when  /// executed.  ArrayRef<Type> FuncOp::getCallableResults() { return getType().getResults(); }  // ....  /// Return the callee of the generic call operation, this is required by the  /// call interface.  CallInterfaceCallable GenericCallOp::getCallableForCallee() {  return getAttrOfType<SymbolRefAttr>("callee");  }  /// Get the argument operands to the called function, this is required by the  /// call interface.  Operation::operand\_range GenericCallOp::getArgOperands() { return inputs(); } |

现在内联器已获知玩具方言，我们可以将 inliner pass 添加到玩具的 pass manager 中：

|  |
| --- |
| pm.addPass(mlir::createInlinerPass()); |

现在让我们看一个工作示例：

|  |
| --- |
| toy.func @multiply\_transpose(%arg0: tensor<\*xf64>, %arg1: tensor<\*xf64>) -> tensor<\*xf64> {  %0 = toy.transpose(%arg0 : tensor<\*xf64>) to tensor<\*xf64>  %1 = toy.transpose(%arg1 : tensor<\*xf64>) to tensor<\*xf64>  %2 = toy.mul %0, %1 : tensor<\*xf64>  toy.return %2 : tensor<\*xf64>  }  toy.func @main() {  %0 = toy.constant dense<[[1.000000e+00, 2.000000e+00, 3.000000e+00], [4.000000e+00, 5.000000e+00, 6.000000e+00]]> : tensor<2x3xf64>  %1 = toy.reshape(%0 : tensor<2x3xf64>) to tensor<2x3xf64>  %2 = toy.constant dense<[1.000000e+00, 2.000000e+00, 3.000000e+00, 4.000000e+00, 5.000000e+00, 6.000000e+00]> : tensor<6xf64>  %3 = toy.reshape(%2 : tensor<6xf64>) to tensor<2x3xf64>  %4 = toy.generic\_call @multiply\_transpose(%1, %3) : (tensor<2x3xf64>, tensor<2x3xf64>) -> tensor<\*xf64>  %5 = toy.generic\_call @multiply\_transpose(%3, %1) : (tensor<2x3xf64>, tensor<2x3xf64>) -> tensor<\*xf64>  toy.print %5 : tensor<\*xf64>  toy.return  } |

我们有两个对 multiply\_transpose 的调用，我们希望将其内联到 main 中，但如果我们查看输出，则没有任何变化。我们遗漏了最后一个微妙的部分：在调用的边缘有一个隐藏的类型转换。如果我们看上面的内容， generic\_call 的操作数是 类型 tensor<2x3xf64>，而函数 需要的输入tensor<\*xf64>。为了解决这个差异，内联器需要插入一个显式转换操作。为此，我们需要向 Toy 方言添加一个新操作ToyCastOp(toy.cast) ，以表示两个不同形状之间的转换。

|  |
| --- |
| def CastOp : Toy\_Op<"cast", [  DeclareOpInterfaceMethods<CastOpInterface>,  NoMemoryEffect,  SameOperandsAndResultShape]  > {  let summary = "shape cast operation";  let description = [{  The "cast" operation converts a tensor from one type to an equivalent type  without changing any data elements. The source and destination types  must both be tensor types with the same element type. If both are ranked,  then shape is required to match. The operation is invalid if converting  to a mismatching constant dimension.  }];  let arguments = (ins F64Tensor:$input);  let results = (outs F64Tensor:$output);  let assemblyFormat = "$input attr-dict `:` type($input) `to` type($output)";  } |

请注意，此转换操作的定义添加一个CastOpInterface到特征列表( traits list)。该接口为类转换操作提供了几个实用程序，例如 folding identity casts 和验证。我们通过为该areCastCompatible方法提供定义来连接到该接口：

|  |
| --- |
| /// Returns true if the given set of input and result types are compatible with  /// this cast operation. This is required by the `CastOpInterface` to verify  /// this operation and provide other additional utilities.  bool CastOp::areCastCompatible(TypeRange inputs, TypeRange outputs) {  if (inputs.size() != 1 || outputs.size() != 1)  return false;  // The inputs must be Tensors with the same element type.  TensorType input = inputs.front().dyn\_cast<TensorType>();  TensorType output = outputs.front().dyn\_cast<TensorType>();  if (!input || !output || input.getElementType() != output.getElementType())  return false;  // The shape is required to match if both types are ranked.  return !input.hasRank() || !output.hasRank() || input == output;  } |

通过适当的转换操作，我们现在可以重载 ToyInlinerInterface 上必要的挂钩，以便在必要时为我们插入它：

|  |
| --- |
| struct ToyInlinerInterface : public DialectInlinerInterface {  ...  /// Attempts to materialize a conversion for a type mismatch between a call  /// from this dialect, and a callable region. This method should generate an  /// operation that takes 'input' as the only operand, and produces a single  /// result of 'resultType'. If a conversion can not be generated, nullptr  /// should be returned.  Operation \*materializeCallConversion(OpBuilder &builder, Value input,  Type resultType,  Location conversionLoc) const final {  return builder.create<CastOp>(conversionLoc, resultType, input);  }  }; |

如果我们再次通过流水线运行工作示例，我们将得到预期的结果：

|  |
| --- |
| toy.func @main() {  %0 = toy.constant dense<[[1.000000e+00, 2.000000e+00, 3.000000e+00], [4.000000e+00, 5.000000e+00, 6.000000e+00]]> : tensor<2x3xf64>  %1 = toy.constant dense<[[1.000000e+00, 2.000000e+00, 3.000000e+00], [4.000000e+00, 5.000000e+00, 6.000000e+00]]> : tensor<2x3xf64>  %2 = toy.cast %1 : tensor<2x3xf64> to tensor<\*xf64>  %3 = toy.cast %0 : tensor<2x3xf64> to tensor<\*xf64>  %4 = toy.transpose(%2 : tensor<\*xf64>) to tensor<\*xf64>  %5 = toy.transpose(%3 : tensor<\*xf64>) to tensor<\*xf64>  %6 = toy.mul %4, %5 : tensor<\*xf64>  toy.print %6 : tensor<\*xf64>  toy.return  } |

注意：通用内联器也会执行简化，因此输出可能比预期的要干净一些。

#### Intraprocedural shape inference

现在我们已经内联了所有函数，剩下的是一个包含静态和动态形状操作混合的主函数。我们现在可以编写一个简单的“ shape inference pass”来在程序内（在单个函数内）传播形状。我们可以将其写成一个pass， 该pass直接对玩具方言中的操作约束进行编码，但这似乎是可以通用编写的转换的良好候选者。作为一个好的经验法则，最好尽可能通用地表达转换，以便将来可以扩展到其他方言。不知道有多少其他方言可能有类似的需求或遇到相同的问题。

对于shape inference，如果我们将问题分解为核心，我们实际上只希望操作告诉我们给定一组静态已知输入的预期输出。（我们肯定可以变得比这更复杂，但为了我们的需要，我们可以保持简单。）鉴于此属性是特定操作的核心，我们可以定义一个操作接口，可以在需要其结果的操作上指定形状推断。

与操作类似，我们也可以 使用操作定义规范 (ODS) 框架来[定义操作接口](https://mlir.llvm.org/docs/Interfaces/" \l "attributeoperationtype-interfaces)。

接口是通过继承OpInterface来定义的，它将要赋予生成的 C++ 接口类的名称作为模板参数。出于我们的目的，我们将简单地命名生成的类ShapeInference。我们还提供了接口的描述。

|  |
| --- |
| def ShapeInferenceOpInterface : OpInterface<"ShapeInference"> {  let description = [{  Interface to access a registered method to infer the return types for an  operation that can be used during type inference.  }];  } |

接下来，我们定义操作需要提供的接口方法。接口方法包括： 描述；字符串形式的 C++ 返回类型；字符串形式的方法名称；和一些可选组件，具体取决于需要。有关详细信息，请参阅 ODS 文档。

|  |
| --- |
| def ShapeInferenceOpInterface : OpInterface<"ShapeInference"> {  ...  let methods = [  InterfaceMethod<"Infer and set the output shape for the current operation.",  "void", "inferShapes">  ];  } |

现在接口已定义，我们可以将其添加到必要的 Toy 操作中，其方式与我们将其添加CallOpInterface到 GenericCallOp 的方式类似：

|  |
| --- |
| def MulOp : Toy\_Op<"mul",  [..., DeclareOpInterfaceMethods<ShapeInferenceOpInterface>]> {  ...  } |

然后，这些操作中的每一个都需要为该 inferShapes()方法提供一个定义。例如，对于乘法操作，结果形状被推断为输入的形状。

|  |
| --- |
| /// Infer the output shape of the MulOp, this is required by the shape inference  /// interface.  void MulOp::inferShapes() { getResult().setType(getOperand(0).getType()); } |

此时，每个必要的 Toy 操作都提供了一种机制来推断其输出形状。ShapeInferencePass 将对函数进行操作：它将在每个函数上独立运行。MLIR 还支持在任何孤立操作上运行通用 OperationPass，但这里我们的模块仅包含函数，因此无需泛化到所有操作。

实现这样的传递是通过创建一个继承自 mlir::OperationPass<FuncOp>并override该runOnOperation()方法的类来完成的。

|  |
| --- |
| class ShapeInferencePass  : public mlir::PassWrapper<ShapeInferencePass, OperationPass<FuncOp>> {  void runOnOperation() override {  FuncOp function = getOperation();  ...  }  }; |

同时，我们还创建一个辅助方法来实例化 pass：

|  |
| --- |
| std::unique\_ptr<mlir::Pass> mlir::toy::createShapeInferencePass() {  return std::make\_unique<ShapeInferencePass>();  } |

形状推断算法的操作如下：

1. 构建一个包含返回动态形状张量的所有操作的工作列表：这些是需要形状推断的操作。
2. 迭代工作列表：

* 找到要处理的操作：工作列表中的下一个就绪操作的所有参数都是非通用的，
* 如果没有找到操作，则跳出循环，
* 从工作列表中删除操作，
* 从参数类型推断其输出的形状。

1. 如果工作列表为空，则算法成功。

在处理上面描述的操作时，我们 ShapeInference使用以下代码片段查询它是否注册了接口：

|  |
| --- |
| // Ask the operation to infer its output shapes.  LLVM\_DEBUG(llvm::dbgs() << "Inferring shape for: " << \*op << "\n");  /// We check if an operation has a particular interface by casting.  if (ShapeInference shapeOp = dyn\_cast<ShapeInference>(op)) {  shapeOp.inferShapes();  } else {  op->emitError("unable to infer shape of operation without shape "  "inference interface");  return signalPassFailure();  } |

然后我们可以将我们的 pass 添加到 pass manager：

|  |
| --- |
| pm.addPass(mlir::createShapeInferencePass()); |

如果我们重新运行我们原来的例子，我们现在得到以下内容：

|  |
| --- |
| toy.func @main() {  %0 = toy.constant dense<[[1.000000e+00, 2.000000e+00, 3.000000e+00], [4.000000e+00, 5.000000e+00, 6.000000e+00]]> : tensor<2x3xf64>  %1 = toy.transpose(%0 : tensor<2x3xf64>) to tensor<3x2xf64>  %2 = toy.mul %1, %1 : tensor<3x2xf64>  toy.print %2 : tensor<3x2xf64>  toy.return  } |

您可以自己构建toyc-ch4并尝试toyc-ch4 test/Examples/Toy/Ch4/codegen.toy -emit=mlir -opt：

在 下一章中，我们将通过针对较低级别的方言来开始代码生成过程，以优化一些计算量更大的 Toy 操作。

## 部分降级到低级方言的优化

https://mlir.llvm.org/docs/Tutorials/Toy/Ch-5/

在这一点上，我们渴望生成实际代码并看到我们的玩具语言获得生命。我们将使用 LLVM 生成代码，但仅在此处显示 LLVM 构建器界面(builder interface)不会很令人兴奋。相反，我们将展示如何通过在同一函数中共存的方言混合来执行渐进式降低。

为了让它更有趣，在本章中，我们将考虑重用方言优化的仿射变换(Affine)中实现的现有优化。这种方言是为程序的计算量大的部分量身定制的，并且是受限的：它不支持表示我们的 toy.print内置函数，例如，也不应该！相反，我们可以针对 Affine，在 Toy 的计算量大的部分，并在 下一章中直接针对 lowering print 的LLVM IR方言 。作为此降低的一部分，我们将从[TensorType](https://mlir.llvm.org/docs/Dialects/Builtin/" \l "rankedtensortype)降低到Toy 操作在 [MemRefType](https://mlir.llvm.org/docs/Dialects/Builtin/" \l "memreftype)，[MemRefType](https://mlir.llvm.org/docs/Dialects/Builtin/" \l "memreftype)是通过仿射循环嵌套索引的。张量表示抽象值类型的数据序列，这意味着它们不存在于任何内存中。另一方面，MemRefs 表示较低级别的缓冲区访问，因为它们是对内存区域的具体引用。

### 方言转换

MLIR 有许多不同的方言，因此有一个统一的框架来在它们之间进行 转换很重要。这就是DialectConversion 框架发挥作用的地方。该框架允许将一组非法操作转换为一组合法操作。要使用这个框架，我们需要提供两件事（以及可选的第三件事）：

* 转换 目标
* 这是对合法转换的操作或方言的正式规范。不合法的操作将需要重写模式来执行[合法化](https://mlir.llvm.org/getting_started/Glossary/" \l "legalization)。
* 一组 重写模式
* 这是一组 用于将 非法操作转换为一组零个或多个合法操作的模式。
* 可选地， 类型转换器。
* 如果提供，这将用于转换块参数的类型。我们的转换不需要这个。

#### 转换目标

出于我们的目的，我们希望将计算密集型Toy操作转换为来自Affine、Arith、Func和MemRef方言的操作组合，以进一步优化。为了开始降低，我们首先定义我们的转换目标：

|  |
| --- |
| void ToyToAffineLoweringPass::runOnOperation() {  // The first thing to define is the conversion target. This will define the  // final target for this lowering.  mlir::ConversionTarget target(getContext());  // We define the specific operations, or dialects, that are legal targets for  // this lowering. In our case, we are lowering to a combination of the  // `Affine`, `Arith`, `Func`, and `MemRef` dialects.  target.addLegalDialect<AffineDialect, arith::ArithDialect,  func::FuncDialect, memref::MemRefDialect>();  // We also define the Toy dialect as Illegal so that the conversion will fail  // if any of these operations are \*not\* converted. Given that we actually want  // a partial lowering, we explicitly mark the Toy operations that don't want  // to lower, `toy.print`, as \*legal\*. `toy.print` will still need its operands  // to be updated though (as we convert from TensorType to MemRefType), so we  // only treat it as `legal` if its operands are legal.  target.addIllegalDialect<ToyDialect>();  target.addDynamicallyLegalOp<toy::PrintOp>([](toy::PrintOp op) {  return llvm::none\_of(op->getOperandTypes(),  [](Type type) { return type.isa<TensorType>(); });  });  ...  } |

上面我们先把toy dialect设置为非法，再把打印操作设置为合法。我们本可以反过来做到这一点。单个操作始终优先于（更通用的）方言定义，因此顺序无关紧要。详情请见ConversionTarget::getOpInfo。

#### 转换模式

定义好转换目标后，我们就可以定义如何将 非法操作转换为合法操作了。与第 3 章介绍的规范化框架类似 ，该 DialectConversion框架也使用 RewritePatterns来执行转换逻辑。这些模式可能是RewritePatterns以前见过的，也可能是特定于转换框架的新型模式ConversionPattern。ConversionPatterns 与传统的RewritePatterns不同之处在于，它们接受额外的operands 参数，包含已重新映射/替换的操作数。这在处理类型转换时使用，因为该模式将希望对新类型的值进行操作但与旧类型进行匹配。对于我们的降低，此不变量将很有用，因为它将 当前正在操作的 [TensorType](https://mlir.llvm.org/docs/Dialects/Builtin/" \l "rankedtensortype)转换为[MemRefType](https://mlir.llvm.org/docs/Dialects/Builtin/" \l "memreftype)。让我们看一下降低toy.transpose操作的片段：

|  |
| --- |
| /// Lower the `toy.transpose` operation to an affine loop nest.  struct TransposeOpLowering : public mlir::ConversionPattern {  TransposeOpLowering(mlir::MLIRContext \*ctx)  : mlir::ConversionPattern(TransposeOp::getOperationName(), 1, ctx) {}  /// Match and rewrite the given `toy.transpose` operation, with the given  /// operands that have been remapped from `tensor<...>` to `memref<...>`.  mlir::LogicalResult  matchAndRewrite(mlir::Operation \*op, ArrayRef<mlir::Value> operands,  mlir::ConversionPatternRewriter &rewriter) const final {  auto loc = op->getLoc();  // Call to a helper function that will lower the current operation to a set  // of affine loops. We provide a functor that operates on the remapped  // operands, as well as the loop induction variables for the inner most  // loop body.  lowerOpToLoops(  op, operands, rewriter,  [loc](mlir::PatternRewriter &rewriter,  ArrayRef<mlir::Value> memRefOperands,  ArrayRef<mlir::Value> loopIvs) {  // Generate an adaptor for the remapped operands of the TransposeOp.  // This allows for using the nice named accessors that are generated  // by the ODS. This adaptor is automatically provided by the ODS  // framework.  TransposeOpAdaptor transposeAdaptor(memRefOperands);  mlir::Value input = transposeAdaptor.input();  // Transpose the elements by generating a load from the reverse  // indices.  SmallVector<mlir::Value, 2> reverseIvs(llvm::reverse(loopIvs));  return rewriter.create<mlir::AffineLoadOp>(loc, input, reverseIvs);  });  return success();  }  }; |

现在我们可以准备在降低过程中使用的模式列表：

|  |
| --- |
| void ToyToAffineLoweringPass::runOnOperation() {  ...  // Now that the conversion target has been defined, we just need to provide  // the set of patterns that will lower the Toy operations.  mlir::RewritePatternSet patterns(&getContext());  patterns.add<..., TransposeOpLowering>(&getContext());  ... |

#### 部分降低

一旦定义了模式，我们就可以执行实际的降低。该 DialectConversion框架提供了几种不同的降低模式，但出于我们的目的，我们将执行部分降低，因为此时我们不会转换 toy.print。

|  |
| --- |
| void ToyToAffineLoweringPass::runOnOperation() {  ...  // With the target and rewrite patterns defined, we can now attempt the  // conversion. The conversion will signal failure if any of our \*illegal\*  // operations were not converted successfully.  if (mlir::failed(mlir::applyPartialConversion(getOperation(), target, patterns)))  signalPassFailure();  } |

**部分降低的设计考虑**

在深入研究我们降低的结果之前，现在是讨论部分降低时潜在设计考虑因素的好时机。在我们的降低中，我们从值类型 TensorType 转换为分配的（类似缓冲区的）类型 MemRefType。但是，鉴于我们不降低 toy.print操作，我们需要暂时弥合这两个世界。有很多方法可以解决这个问题，每种方法都有自己的权衡：

* 从缓冲区生成load操作
* 一种选择是从缓冲区类型生成load操作以具体化值类型的实例。这允许toy.print操作的定义保持不变。这种方法的缺点是对affine方言的优化是有限的，因为 load 实际上涉及一个完整的副本，只有在我们的优化执行后才可见。
* 生成在降低的类型上运行的新版本toy.print
* 另一种选择是让另一个降低的变体toy.print 在降低的类型上运行。这个选项的好处是没有隐藏的、不必要的复制到优化器。缺点是需要另一个操作定义，它可能会复制第一个操作的许多方面。在ODS中定义一个基类 可能会简化这一点，但您仍然需要分别对待这些操作。
* 更新toy.print以允许在降低的类型上操作
* 第三个选项是更新当前的定义toy.print以允许在降低的类型上运行。这种方式的好处是简单，不会引入额外的隐藏副本，也不需要另外定义操作。此选项的缺点是它需要在Toy方言中混合抽象级别。

为了简单起见，我们将使用第三个选项来进行此降低。这涉及更新操作定义文件中 PrintOp 的类型约束：

|  |
| --- |
| def PrintOp : Toy\_Op<"print"> {  ...  // The print operation takes an input tensor to print.  // We also allow a F64MemRef to enable interop during partial lowering.  let arguments = (ins AnyTypeOf<[F64Tensor, F64MemRef]>:$input);  } |

### 完整的玩具示例

让我们举一个具体的例子：

|  |
| --- |
| toy.func @main() {  %0 = toy.constant dense<[[1.000000e+00, 2.000000e+00, 3.000000e+00], [4.000000e+00, 5.000000e+00, 6.000000e+00]]> : tensor<2x3xf64>  %2 = toy.transpose(%0 : tensor<2x3xf64>) to tensor<3x2xf64>  %3 = toy.mul %2, %2 : tensor<3x2xf64>  toy.print %3 : tensor<3x2xf64>  toy.return  } |

通过将仿射降低添加到我们的管道中，我们现在可以生成：

|  |
| --- |
| func.func @main() {  %cst = arith.constant 1.000000e+00 : f64  %cst\_0 = arith.constant 2.000000e+00 : f64  %cst\_1 = arith.constant 3.000000e+00 : f64  %cst\_2 = arith.constant 4.000000e+00 : f64  %cst\_3 = arith.constant 5.000000e+00 : f64  %cst\_4 = arith.constant 6.000000e+00 : f64  // Allocating buffers for the inputs and outputs.  %0 = memref.alloc() : memref<3x2xf64>  %1 = memref.alloc() : memref<3x2xf64>  %2 = memref.alloc() : memref<2x3xf64>  // Initialize the input buffer with the constant values.  affine.store %cst, %2[0, 0] : memref<2x3xf64>  affine.store %cst\_0, %2[0, 1] : memref<2x3xf64>  affine.store %cst\_1, %2[0, 2] : memref<2x3xf64>  affine.store %cst\_2, %2[1, 0] : memref<2x3xf64>  affine.store %cst\_3, %2[1, 1] : memref<2x3xf64>  affine.store %cst\_4, %2[1, 2] : memref<2x3xf64>  // Load the transpose value from the input buffer and store it into the  // next input buffer.  affine.for %arg0 = 0 to 3 {  affine.for %arg1 = 0 to 2 {  %3 = affine.load %2[%arg1, %arg0] : memref<2x3xf64>  affine.store %3, %1[%arg0, %arg1] : memref<3x2xf64>  }  }  // Multiply and store into the output buffer.  affine.for %arg0 = 0 to 3 {  affine.for %arg1 = 0 to 2 {  %3 = affine.load %1[%arg0, %arg1] : memref<3x2xf64>  %4 = affine.load %1[%arg0, %arg1] : memref<3x2xf64>  %5 = arith.mulf %3, %4 : f64  affine.store %5, %0[%arg0, %arg1] : memref<3x2xf64>  }  }  // Print the value held by the buffer.  toy.print %0 : memref<3x2xf64>  memref.dealloc %2 : memref<2x3xf64>  memref.dealloc %1 : memref<3x2xf64>  memref.dealloc %0 : memref<3x2xf64>  return  } |

### 利用仿射优化

我们简单的降低是正确的，但在效率方面还有很多不足之处。例如，降低toy.mul产生了一些冗余负载。让我们看看如何向管道添加一些现有的优化可以帮助清理它。将LoopFusion和MemRefDataFlowOpt传递到管道会得到以下结果：

|  |
| --- |
| func.func @main() {  %cst = arith.constant 1.000000e+00 : f64  %cst\_0 = arith.constant 2.000000e+00 : f64  %cst\_1 = arith.constant 3.000000e+00 : f64  %cst\_2 = arith.constant 4.000000e+00 : f64  %cst\_3 = arith.constant 5.000000e+00 : f64  %cst\_4 = arith.constant 6.000000e+00 : f64  // Allocating buffers for the inputs and outputs.  %0 = memref.alloc() : memref<3x2xf64>  %1 = memref.alloc() : memref<2x3xf64>  // Initialize the input buffer with the constant values.  affine.store %cst, %1[0, 0] : memref<2x3xf64>  affine.store %cst\_0, %1[0, 1] : memref<2x3xf64>  affine.store %cst\_1, %1[0, 2] : memref<2x3xf64>  affine.store %cst\_2, %1[1, 0] : memref<2x3xf64>  affine.store %cst\_3, %1[1, 1] : memref<2x3xf64>  affine.store %cst\_4, %1[1, 2] : memref<2x3xf64>  affine.for %arg0 = 0 to 3 {  affine.for %arg1 = 0 to 2 {  // Load the transpose value from the input buffer.  %2 = affine.load %1[%arg1, %arg0] : memref<2x3xf64>  // Multiply and store into the output buffer.  %3 = arith.mulf %2, %2 : f64  affine.store %3, %0[%arg0, %arg1] : memref<3x2xf64>  }  }  // Print the value held by the buffer.  toy.print %0 : memref<3x2xf64>  memref.dealloc %1 : memref<2x3xf64>  memref.dealloc %0 : memref<3x2xf64>  return  } |

在这里，我们可以看到去掉了一个多余的分配，融合了两个循环嵌套，去掉了一些不需要的loads。您可以自己构建toyc-ch5 并尝试toyc-ch5 test/Examples/Toy/Ch5/affine-lowering.mlir -emit=mlir-affine：我们还可以通过添加来检查我们的优化-opt。

在本章中，我们探讨了部分降低的某些方面，目的是进行优化。在 下一章中，我们将通过针对 LLVM 进行代码生成来继续讨论方言转换。

## 降低到 LLVM 和代码生成

在 上一章中，我们介绍了 [方言转换](https://mlir.llvm.org/docs/DialectConversion/) 框架，并将很多Toy操作部分降低到仿射循环嵌套进行优化。在本章中，我们将最终降低到 LLVM 进行代码生成。

### 降低到 LLVM

对于这种降低，我们将再次使用方言转换框架来执行繁重的工作。然而，这一次，我们将完全转换为 [LLVM 方言](https://mlir.llvm.org/docs/Dialects/LLVM/)。值得庆幸的是，我们已经降低了除一个操作之外的所有toy操作，最后一个是toy.print。在继续转换为 LLVM 之前，让我们降低toy.print操作。我们将此操作降低为针对每个元素调用printf的非仿射循环嵌套。注意，因为方言转换框架支持 [传递降低](https://mlir.llvm.org/getting_started/Glossary/" \l "transitive-lowering)，我们不需要直接在 LLVM 方言中发出操作。通过传递降低，我们的意思是，转换框架可以应用多种模式来完全合法化操作。在此示例中，我们生成结构化循环嵌套而不是 LLVM 方言中的分支形式。只要我们从循环操作降低到 LLVM，降低仍然会成功。

在降低期间，我们可以获得或构建 printf 的声明，如下所示：

|  |
| --- |
| /// Return a symbol reference to the printf function, inserting it into the  /// module if necessary.  static FlatSymbolRefAttr getOrInsertPrintf(PatternRewriter &rewriter,  ModuleOp module,  LLVM::LLVMDialect \*llvmDialect) {  auto \*context = module.getContext();  if (module.lookupSymbol<LLVM::LLVMFuncOp>("printf"))  return SymbolRefAttr::get("printf", context);  // Create a function declaration for printf, the signature is:  // \* `i32 (i8\*, ...)`  auto llvmI32Ty = IntegerType::get(context, 32);  auto llvmI8PtrTy =  LLVM::LLVMPointerType::get(IntegerType::get(context, 8));  auto llvmFnType = LLVM::LLVMFunctionType::get(llvmI32Ty, llvmI8PtrTy,  /\*isVarArg=\*/true);  // Insert the printf function into the body of the parent module.  PatternRewriter::InsertionGuard insertGuard(rewriter);  rewriter.setInsertionPointToStart(module.getBody());  rewriter.create<LLVM::LLVMFuncOp>(module.getLoc(), "printf", llvmFnType);  return SymbolRefAttr::get("printf", context);  } |

现在已经定义了 printf 操作的降低，我们可以指定降低所需的组件。这些与[前一章](https://mlir.llvm.org/docs/Tutorials/Toy/Ch-5/)中定义的组件基本相同 。

#### 转换目标

对于此转换，除了顶级模块之外，我们将把所有内容都降低到 LLVM 方言。

|  |
| --- |
| mlir::ConversionTarget target(getContext());  target.addLegalDialect<mlir::LLVMDialect>();  target.addLegalOp<mlir::ModuleOp>(); |

#### 类型转换器

这种降低也会将当前正在操作的 MemRef 类型转换为 LLVM 中的表示。为了执行此转换，我们使用 TypeConverter 作为降低的一部分。此转换器指定一种类型如何映射到另一种类型。现在这是必要的，因为我们正在执行涉及块参数的更复杂的降低。鉴于我们没有任何需要降低的特定于 Toy 方言的类型，默认转换器就足以满足我们的用例。

|  |
| --- |
| LLVMTypeConverter typeConverter(&getContext()); |

#### 转换模式

现在已经定义了转换目标，我们需要提供用于降低的模式。在编译过程的这一点上，我们有toy、affine、arith和std操作的组合。幸运的是 affine，arith、 和std方言已经提供了将它们转换为 LLVM 方言所需的一组模式。这些模式允许通过依赖[传递降低](https://mlir.llvm.org/getting_started/Glossary/" \l "transitive-lowering)来在多个阶段降低 IR 。

|  |
| --- |
| mlir::RewritePatternSet patterns(&getContext());  mlir::populateAffineToStdConversionPatterns(patterns, &getContext());  mlir::cf::populateSCFToControlFlowConversionPatterns(patterns, &getContext());  mlir::arith::populateArithToLLVMConversionPatterns(typeConverter,  patterns);  mlir::populateFuncToLLVMConversionPatterns(typeConverter, patterns);  mlir::cf::populateControlFlowToLLVMConversionPatterns(patterns, &getContext());  // The only remaining operation, to lower from the `toy` dialect, is the  // PrintOp.  patterns.add<PrintOpLowering>(&getContext()); |

#### 完全下降

我们想完全降低到 LLVM，所以我们使用FullConversion. 这可确保在转换后仅保留合法操作。

|  |
| --- |
| mlir::ModuleOp module = getOperation();  if (mlir::failed(mlir::applyFullConversion(module, target, patterns)))  signalPassFailure(); |

回顾我们当前的工作示例：

|  |
| --- |
| toy.func @main() {  %0 = toy.constant dense<[[1.000000e+00, 2.000000e+00, 3.000000e+00], [4.000000e+00, 5.000000e+00, 6.000000e+00]]> : tensor<2x3xf64>  %2 = toy.transpose(%0 : tensor<2x3xf64>) to tensor<3x2xf64>  %3 = toy.mul %2, %2 : tensor<3x2xf64>  toy.print %3 : tensor<3x2xf64>  toy.return  } |

我们现在可以深入到 LLVM 方言，它会生成以下代码：

|  |
| --- |
| llvm.func @free(!llvm<"i8\*">)  llvm.func @printf(!llvm<"i8\*">, ...) -> i32  llvm.func @malloc(i64) -> !llvm<"i8\*">  llvm.func @main() {  %0 = llvm.mlir.constant(1.000000e+00 : f64) : f64  %1 = llvm.mlir.constant(2.000000e+00 : f64) : f64  ...  ^bb16:  %221 = llvm.extractvalue %25[0] : !llvm<"{ double\*, i64, [2 x i64], [2 x i64] }">  %222 = llvm.mlir.constant(0 : index) : i64  %223 = llvm.mlir.constant(2 : index) : i64  %224 = llvm.mul %214, %223 : i64  %225 = llvm.add %222, %224 : i64  %226 = llvm.mlir.constant(1 : index) : i64  %227 = llvm.mul %219, %226 : i64  %228 = llvm.add %225, %227 : i64  %229 = llvm.getelementptr %221[%228] : (!llvm."double\*">, i64) -> !llvm<"f64\*">  %230 = llvm.load %229 : !llvm<"double\*">  %231 = llvm.call @printf(%207, %230) : (!llvm<"i8\*">, f64) -> i32  %232 = llvm.add %219, %218 : i64  llvm.br ^bb15(%232 : i64)  ...  ^bb18:  %235 = llvm.extractvalue %65[0] : !llvm<"{ double\*, i64, [2 x i64], [2 x i64] }">  %236 = llvm.bitcast %235 : !llvm<"double\*"> to !llvm<"i8\*">  llvm.call @free(%236) : (!llvm<"i8\*">) -> ()  %237 = llvm.extractvalue %45[0] : !llvm<"{ double\*, i64, [2 x i64], [2 x i64] }">  %238 = llvm.bitcast %237 : !llvm<"double\*"> to !llvm<"i8\*">  llvm.call @free(%238) : (!llvm<"i8\*">) -> ()  %239 = llvm.extractvalue %25[0] : !llvm<"{ double\*, i64, [2 x i64], [2 x i64] }">  %240 = llvm.bitcast %239 : !llvm<"double\*"> to !llvm<"i8\*">  llvm.call @free(%240) : (!llvm<"i8\*">) -> ()  llvm.return  } |

有关降低到 LLVM 方言的更深入的详细信息，请参阅 [LLVM IR Target](https://mlir.llvm.org/docs/TargetLLVMIR/) 。

### CodeGen：摆脱 MLIR

此时我们正处于代码生成的风口浪尖。我们可以用 LLVM 方言生成代码，所以现在我们只需要导出到 LLVM IR 并设置 JIT 来运行它。

#### emitting LLVM IR

现在我们的模块只包含 LLVM 方言中的操作，我们可以导出到 LLVM IR。要以编程方式执行此操作，我们可以调用以下实用程序：

|  |
| --- |
| std::unique\_ptr<llvm::Module> llvmModule = mlir::translateModuleToLLVMIR(module);  if (!llvmModule)  /\* ... an error was encountered ... \*/ |

将我们的模块导出到 LLVM IR 会生成：

|  |
| --- |
| define void @main() {  ...  102:  %103 = extractvalue { double\*, i64, [2 x i64], [2 x i64] } %8, 0  %104 = mul i64 %96, 2  %105 = add i64 0, %104  %106 = mul i64 %100, 1  %107 = add i64 %105, %106  %108 = getelementptr double, double\* %103, i64 %107  %109 = memref.load double, double\* %108  %110 = call i32 (i8\*, ...) @printf(i8\* getelementptr inbounds ([4 x i8], [4 x i8]\* @frmt\_spec, i64 0, i64 0), double %109)  %111 = add i64 %100, 1  cf.br label %99  ...  115:  %116 = extractvalue { double\*, i64, [2 x i64], [2 x i64] } %24, 0  %117 = bitcast double\* %116 to i8\*  call void @free(i8\* %117)  %118 = extractvalue { double\*, i64, [2 x i64], [2 x i64] } %16, 0  %119 = bitcast double\* %118 to i8\*  call void @free(i8\* %119)  %120 = extractvalue { double\*, i64, [2 x i64], [2 x i64] } %8, 0  %121 = bitcast double\* %120 to i8\*  call void @free(i8\* %121)  ret void  } |

如果我们在生成的 LLVM IR 上启用优化，我们可以将其减少很多：

|  |
| --- |
| define void @main()  %0 = tail call i32 (i8\*, ...) @printf(i8\* nonnull dereferenceable(1) getelementptr inbounds ([4 x i8], [4 x i8]\* @frmt\_spec, i64 0, i64 0), double 1.000000e+00)  %1 = tail call i32 (i8\*, ...) @printf(i8\* nonnull dereferenceable(1) getelementptr inbounds ([4 x i8], [4 x i8]\* @frmt\_spec, i64 0, i64 0), double 1.600000e+01)  %putchar = tail call i32 @putchar(i32 10)  %2 = tail call i32 (i8\*, ...) @printf(i8\* nonnull dereferenceable(1) getelementptr inbounds ([4 x i8], [4 x i8]\* @frmt\_spec, i64 0, i64 0), double 4.000000e+00)  %3 = tail call i32 (i8\*, ...) @printf(i8\* nonnull dereferenceable(1) getelementptr inbounds ([4 x i8], [4 x i8]\* @frmt\_spec, i64 0, i64 0), double 2.500000e+01)  %putchar.1 = tail call i32 @putchar(i32 10)  %4 = tail call i32 (i8\*, ...) @printf(i8\* nonnull dereferenceable(1) getelementptr inbounds ([4 x i8], [4 x i8]\* @frmt\_spec, i64 0, i64 0), double 9.000000e+00)  %5 = tail call i32 (i8\*, ...) @printf(i8\* nonnull dereferenceable(1) getelementptr inbounds ([4 x i8], [4 x i8]\* @frmt\_spec, i64 0, i64 0), double 3.600000e+01)  %putchar.2 = tail call i32 @putchar(i32 10)  ret void  } |

可以 examples/toy/Ch6/toy.cpp在dumpLLVMIR()函数中找到用于转储 LLVM IR 的完整代码清单：

|  |
| --- |
| int dumpLLVMIR(mlir::ModuleOp module) {  // Translate the module, that contains the LLVM dialect, to LLVM IR. Use a  // fresh LLVM IR context. (Note that LLVM is not thread-safe and any  // concurrent use of a context requires external locking.)  llvm::LLVMContext llvmContext;  auto llvmModule = mlir::translateModuleToLLVMIR(module, llvmContext);  if (!llvmModule) {  llvm::errs() << "Failed to emit LLVM IR\n";  return -1;  }  // Initialize LLVM targets.  llvm::InitializeNativeTarget();  llvm::InitializeNativeTargetAsmPrinter();  mlir::ExecutionEngine::setupTargetTriple(llvmModule.get());  /// Optionally run an optimization pipeline over the llvm module.  auto optPipeline = mlir::makeOptimizingTransformer(  /\*optLevel=\*/EnableOpt ? 3 : 0, /\*sizeLevel=\*/0,  /\*targetMachine=\*/nullptr);  if (auto err = optPipeline(llvmModule.get())) {  llvm::errs() << "Failed to optimize LLVM IR " << err << "\n";  return -1;  }  llvm::errs() << \*llvmModule << "\n";  return 0;  } |

#### 设置 JIT

可以使用mlir::ExecutionEngine基础架构设置 JIT 来运行包含 LLVM 方言的模块。这是一个围绕 LLVM 的 JIT 接受.mlir输入的实用程序包装器。可以Ch6/toyc.cpp在runJit()函数中找到用于设置 JIT 的完整代码清单：

|  |
| --- |
| int runJit(mlir::ModuleOp module) {  // Initialize LLVM targets.  llvm::InitializeNativeTarget();  llvm::InitializeNativeTargetAsmPrinter();  // An optimization pipeline to use within the execution engine.  auto optPipeline = mlir::makeOptimizingTransformer(  /\*optLevel=\*/EnableOpt ? 3 : 0, /\*sizeLevel=\*/0,  /\*targetMachine=\*/nullptr);  // Create an MLIR execution engine. The execution engine eagerly JIT-compiles  // the module.  auto maybeEngine = mlir::ExecutionEngine::create(module,  /\*llvmModuleBuilder=\*/nullptr, optPipeline);  assert(maybeEngine && "failed to construct an execution engine");  auto &engine = maybeEngine.get();  // Invoke the JIT-compiled function.  auto invocationResult = engine->invoke("main");  if (invocationResult) {  llvm::errs() << "JIT invocation failed\n";  return -1;  }  return 0;  } |

您可以从构建目录中使用它：

|  |
| --- |
| $ echo 'def main() { print([[1, 2], [3, 4]]); }' | ./bin/toyc-ch6 -emit=jit  1.000000 2.000000  3.000000 4.000000 |

您还可以使用-emit=mlir、-emit=mlir-affine、-emit=mlir-llvm和 -emit=llvm来比较所涉及的各种 IR 级别。还可以尝试 [--mlir-print-ir-after-all](https://mlir.llvm.org/docs/PassManagement/" \l "ir-printing)跟踪 IR 在整个管道中的演变等选项。

本节中使用的示例代码可以在 test/Examples/Toy/Ch6/llvm-lowering.mlir 中找到。

到目前为止，我们已经使用了原始数据类型。在 [下一章](https://mlir.llvm.org/docs/Tutorials/Toy/Ch-7/)中，我们将添加一个复合struct类型。

## 为玩具添加复合类型

在 上一章中，我们演示了从 Toy 前端到 LLVM IR 的端到端编译流程。在本章中，我们将扩展 Toy 语言以支持新的复合struct类型。

### 在玩具中 定义 struct

我们需要定义的第一件事是在我们的toy 源语言中这种类型的接口。Toy 中struct类型的一般语法如下：

|  |
| --- |
| # A struct is defined by using the `struct` keyword followed by a name.  struct MyStruct {  # Inside of the struct is a list of variable declarations without initializers  # or shapes, which may also be other previously defined structs.  var a;  var b;  } |

结构现在可以在函数中用作变量或参数，方法是使用struct的名称而不是var。通过‘.’访问运算符访问struct的成员。struct类型的值可以用复合初始化器或由{}包起来的其它初始化器的逗号分隔的列表。一个例子如下所示：

|  |
| --- |
| struct Struct {  var a;  var b;  }  # User defined generic function may operate on struct types as well.  def multiply\_transpose(Struct value) {  # We can access the elements of a struct via the '.' operator.  return transpose(value.a) \* transpose(value.b);  }  def main() {  # We initialize struct values using a composite initializer.  Struct value = {[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]};  # We pass these arguments to functions like we do with variables.  var c = multiply\_transpose(value);  print(c);  } |

### struct在 MLIR 中 定义

在 MLIR 中，我们还需要struct类型的表示。MLIR 不提供完全满足我们需要的类型，因此我们需要定义自己的类型。我们将简单地将我们定义struct为一组元素类型的未命名容器。the 的名称struct及其元素仅对我们toy编译器的 AST 有用，因此我们不需要将其编码为 MLIR 表示形式。

#### 定义类型类

如 第 2 章所述， MLIR 中的Type 对象是值类型的，并且依赖于拥有一个内部存储对象来保存该类型的实际数据。该Type类本身充当内部TypeStorage对象的简单包装器，该内部TypeStorage对象对应于MLIRContext的一个实例。构造 一个Type时，我们只是在内部构造和唯一化存储类的一个实例。

当定义一个新的Type，该Type包含参数数据（例如struct 类型，它需要额外的信息来保存元素类型），我们需要提供一个派生存储类。singleton 类型没有任何附加数据的类型（例如[index type](https://mlir.llvm.org/docs/Dialects/Builtin/" \l "indextype)），因此，不需要存储类并使用默认的TypeStorage.

##### 定义存储类

类型存储对象包含构造和唯一化类型实例所需的所有数据。派生存储类必须继承自基类mlir::TypeStorage， 并提供一组别名和挂钩，供MLIRContext 进行唯一化使用。下面是我们struct类型的存储实例的定义，其中详细说明了每个必要的要求：

|  |
| --- |
| /// This class represents the internal storage of the Toy `StructType`.  struct StructTypeStorage : public mlir::TypeStorage {  /// The `KeyTy` is a required type that provides an interface for the storage  /// instance. This type will be used when uniquing an instance of the type  /// storage. For our struct type, we will unique each instance structurally on  /// the elements that it contains.  using KeyTy = llvm::ArrayRef<mlir::Type>;  /// A constructor for the type storage instance.  StructTypeStorage(llvm::ArrayRef<mlir::Type> elementTypes)  : elementTypes(elementTypes) {}  /// Define the comparison function for the key type with the current storage  /// instance. This is used when constructing a new instance to ensure that we  /// haven't already uniqued an instance of the given key.  bool operator==(const KeyTy &key) const { return key == elementTypes; }  /// Define a hash function for the key type. This is used when uniquing  /// instances of the storage.  /// Note: This method isn't necessary as both llvm::ArrayRef and mlir::Type  /// have hash functions available, so we could just omit this entirely.  static llvm::hash\_code hashKey(const KeyTy &key) {  return llvm::hash\_value(key);  }  /// Define a construction function for the key type from a set of parameters.  /// These parameters will be provided when constructing the storage instance  /// itself, see the `StructType::get` method further below.  /// Note: This method isn't necessary because KeyTy can be directly  /// constructed with the given parameters.  static KeyTy getKey(llvm::ArrayRef<mlir::Type> elementTypes) {  return KeyTy(elementTypes);  }  /// Define a construction method for creating a new instance of this storage.  /// This method takes an instance of a storage allocator, and an instance of a  /// `KeyTy`. The given allocator must be used for \*all\* necessary dynamic  /// allocations used to create the type storage and its internal.  static StructTypeStorage \*construct(mlir::TypeStorageAllocator &allocator,  const KeyTy &key) {  // Copy the elements from the provided `KeyTy` into the allocator.  llvm::ArrayRef<mlir::Type> elementTypes = allocator.copyInto(key);  // Allocate the storage instance and construct it.  return new (allocator.allocate<StructTypeStorage>())  StructTypeStorage(elementTypes);  }  /// The following field contains the element types of the struct.  llvm::ArrayRef<mlir::Type> elementTypes;  }; |

##### 定义类型类

定义存储类后，我们可以添加用户可见 StructType类的定义。这是我们实际要与之交互的类。

ToyDialect我们以与处理操作类似的方式在初始化程序中注册此类型：

|  |
| --- |
| /// This class defines the Toy struct type. It represents a collection of  /// element types. All derived types in MLIR must inherit from the CRTP class  /// 'Type::TypeBase'. It takes as template parameters the concrete type  /// (StructType), the base class to use (Type), and the storage class  /// (StructTypeStorage).  class StructType : public mlir::Type::TypeBase<StructType, mlir::Type,  StructTypeStorage> {  public:  /// Inherit some necessary constructors from 'TypeBase'.  using Base::Base;  /// Create an instance of a `StructType` with the given element types. There  /// \*must\* be at least one element type.  static StructType get(llvm::ArrayRef<mlir::Type> elementTypes) {  assert(!elementTypes.empty() && "expected at least 1 element type");  // Call into a helper 'get' method in 'TypeBase' to get a uniqued instance  // of this type. The first parameter is the context to unique in. The  // parameters after are forwarded to the storage instance.  mlir::MLIRContext \*ctx = elementTypes.front().getContext();  return Base::get(ctx, elementTypes);  }  /// Returns the element types of this struct type.  llvm::ArrayRef<mlir::Type> getElementTypes() {  // 'getImpl' returns a pointer to the internal storage instance.  return getImpl()->elementTypes;  }  /// Returns the number of element type held by this struct.  size\_t getNumElementTypes() { return getElementTypes().size(); }  }; |

我们以与处理操作类似的方式在 ToyDialect 初始化程序中注册此类型：

|  |
| --- |
| void ToyDialect::initialize() {  addTypes<StructType>();  } |

（这里需要注意的是，在注册类型时，存储类的定义必须是可见的。）

有了这个，我们现在可以StructType在从 Toy 生成 MLIR 时使用我们的。有关详细信息，请参阅 examples/toy/Ch7/mlir/MLIRGen.cpp。

#### Exposing to ODS

定义新类型后，我们应该让 ODS 框架知道我们的类型，以便我们可以在操作定义中使用它，并在方言中自动生成实用程序。一个简单的例子如下所示：

|  |
| --- |
| // Provide a definition for the Toy StructType for use in ODS. This allows for  // using StructType in a similar way to Tensor or MemRef. We use `DialectType`  // to demarcate the StructType as belonging to the Toy dialect.  def Toy\_StructType :  DialectType<Toy\_Dialect, CPred<"$\_self.isa<StructType>()">,  "Toy struct type">;  // Provide a definition of the types that are used within the Toy dialect.  def Toy\_Type : AnyTypeOf<[F64Tensor, Toy\_StructType]>; |

### 解析和打印

此时，在MLIR 生成和转换期间，我们可以使用我们StructType; 但我们不能输出或解析.mlir。为此，我们需要添加对解析和打印StructType的实例的支持. 这可以通过重写 ToyDialect 上的parseType和printType方法来完成。如上一节所述，当类型公开给 ODS 时，将自动提供这些方法的声明。

|  |
| --- |
| class ToyDialect : public mlir::Dialect {  public:  /// Parse an instance of a type registered to the toy dialect.  mlir::Type parseType(mlir::DialectAsmParser &parser) const override;  /// Print an instance of a type registered to the toy dialect.  void printType(mlir::Type type,  mlir::DialectAsmPrinter &printer) const override;  }; |

这些方法采用高级解析器或printer的一个实例，可以轻松实现必要的功能。在进入实现之前，让我们考虑一下struct 打印 IR 中的类型所需的语法。如 [MLIR language reference](https://mlir.llvm.org/docs/LangRef/" \l "dialect-types)中所述，方言类型通常表示为：! dialect-namespace < type-data >，在某些情况下可以使用漂亮的形式。我们Toy 的解析器和打印机的责任是提供type-data bits。我们将定义我们 的StructType为具有以下形式：

|  |
| --- |
| struct-type ::= `struct` `<` type (`,` type)\* `>` |

#### 解析

解析器的实现如下所示：

|  |
| --- |
| /// Parse an instance of a type registered to the toy dialect.  mlir::Type ToyDialect::parseType(mlir::DialectAsmParser &parser) const {  // Parse a struct type in the following form:  // struct-type ::= `struct` `<` type (`,` type)\* `>`  // NOTE: All MLIR parser function return a ParseResult. This is a  // specialization of LogicalResult that auto-converts to a `true` boolean  // value on failure to allow for chaining, but may be used with explicit  // `mlir::failed/mlir::succeeded` as desired.  // Parse: `struct` `<`  if (parser.parseKeyword("struct") || parser.parseLess())  return Type();  // Parse the element types of the struct.  SmallVector<mlir::Type, 1> elementTypes;  do {  // Parse the current element type.  SMLoc typeLoc = parser.getCurrentLocation();  mlir::Type elementType;  if (parser.parseType(elementType))  return nullptr;  // Check that the type is either a TensorType or another StructType.  if (!elementType.isa<mlir::TensorType, StructType>()) {  parser.emitError(typeLoc, "element type for a struct must either "  "be a TensorType or a StructType, got: ")  << elementType;  return Type();  }  elementTypes.push\_back(elementType);  // Parse the optional: `,`  } while (succeeded(parser.parseOptionalComma()));  // Parse: `>`  if (parser.parseGreater())  return Type();  return StructType::get(elementTypes);  } |

#### print

printer的实现如下所示：

|  |
| --- |
| /// Print an instance of a type registered to the toy dialect.  void ToyDialect::printType(mlir::Type type,  mlir::DialectAsmPrinter &printer) const {  // Currently the only toy type is a struct type.  StructType structType = type.cast<StructType>();  // Print the struct type according to the parser format.  printer << "struct<";  llvm::interleaveComma(structType.getElementTypes(), printer);  printer << '>';  } |

在继续之前，让我们看一个快速示例，展示我们现在拥有的功能：

|  |
| --- |
| struct Struct {  var a;  var b;  }  def multiply\_transpose(Struct value) {  } |

生成以下内容：

|  |
| --- |
| module {  toy.func @multiply\_transpose(%arg0: !toy.struct<tensor<\*xf64>, tensor<\*xf64>>) {  toy.return  }  } |

### Operating on StructType

现在struct已经定义了类型，我们可以通过 IR 来回传递它。下一步是增加对在我们的操作中使用它的支持。

#### 更新现有操作

我们现有的一些操作（例如ReturnOp）将需要更新以处理Toy\_StructType.

|  |
| --- |
| def ReturnOp : Toy\_Op<"return", [Terminator, HasParent<"FuncOp">]> {  ...  let arguments = (ins Variadic<Toy\_Type>:$input);  ...  } |

#### 添加新Toy操作

除了现有操作之外，我们还将添加一些新操作，以提供对structs 的更具体的处理.

**toy.struct\_constant**

这个新操作实现了结构的常量值。在我们当前的建模中，我们只使用一个 数组属性 ，其中包含每个struct元素的一组常量值。

|  |
| --- |
| %0 = toy.struct\_constant [  dense<[[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]]> : tensor<2x3xf64>  ] : !toy.struct<tensor<\*xf64>> |

**toy.struct\_access**

这个新操作具体化了struct值的第 N 个元素。

|  |
| --- |
| // Using %0 from above  %1 = toy.struct\_access %0[0] : !toy.struct<tensor<\*xf64>> -> tensor<\*xf64> |

通过这些操作，我们可以重新审视我们原来的例子：

|  |
| --- |
| struct Struct {  var a;  var b;  }  # User defined generic function may operate on struct types as well.  def multiply\_transpose(Struct value) {  # We can access the elements of a struct via the '.' operator.  return transpose(value.a) \* transpose(value.b);  }  def main() {  # We initialize struct values using a composite initializer.  Struct value = {[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]};  # We pass these arguments to functions like we do with variables.  var c = multiply\_transpose(value);  print(c);  } |

最后得到一个完整的 MLIR 模块：

|  |
| --- |
| module {  toy.func @multiply\_transpose(%arg0: !toy.struct<tensor<\*xf64>, tensor<\*xf64>>) -> tensor<\*xf64> {  %0 = toy.struct\_access %arg0[0] : !toy.struct<tensor<\*xf64>, tensor<\*xf64>> -> tensor<\*xf64>  %1 = toy.transpose(%0 : tensor<\*xf64>) to tensor<\*xf64>  %2 = toy.struct\_access %arg0[1] : !toy.struct<tensor<\*xf64>, tensor<\*xf64>> -> tensor<\*xf64>  %3 = toy.transpose(%2 : tensor<\*xf64>) to tensor<\*xf64>  %4 = toy.mul %1, %3 : tensor<\*xf64>  toy.return %4 : tensor<\*xf64>  }  toy.func @main() {  %0 = toy.struct\_constant [  dense<[[1.000000e+00, 2.000000e+00, 3.000000e+00], [4.000000e+00, 5.000000e+00, 6.000000e+00]]> : tensor<2x3xf64>,  dense<[[1.000000e+00, 2.000000e+00, 3.000000e+00], [4.000000e+00, 5.000000e+00, 6.000000e+00]]> : tensor<2x3xf64>  ] : !toy.struct<tensor<\*xf64>, tensor<\*xf64>>  %1 = toy.generic\_call @multiply\_transpose(%0) : (!toy.struct<tensor<\*xf64>, tensor<\*xf64>>) -> tensor<\*xf64>  toy.print %1 : tensor<\*xf64>  toy.return  }  } |

#### 优化操作StructType

现在我们在 上进行了一些操作StructType，我们也有许多新的常量折叠机会。

内联后，上一节中的 MLIR 模块类似于：

|  |
| --- |
| module {  toy.func @main() {  %0 = toy.struct\_constant [  dense<[[1.000000e+00, 2.000000e+00, 3.000000e+00], [4.000000e+00, 5.000000e+00, 6.000000e+00]]> : tensor<2x3xf64>,  dense<[[1.000000e+00, 2.000000e+00, 3.000000e+00], [4.000000e+00, 5.000000e+00, 6.000000e+00]]> : tensor<2x3xf64>  ] : !toy.struct<tensor<\*xf64>, tensor<\*xf64>>  %1 = toy.struct\_access %0[0] : !toy.struct<tensor<\*xf64>, tensor<\*xf64>> -> tensor<\*xf64>  %2 = toy.transpose(%1 : tensor<\*xf64>) to tensor<\*xf64>  %3 = toy.struct\_access %0[1] : !toy.struct<tensor<\*xf64>, tensor<\*xf64>> -> tensor<\*xf64>  %4 = toy.transpose(%3 : tensor<\*xf64>) to tensor<\*xf64>  %5 = toy.mul %2, %4 : tensor<\*xf64>  toy.print %5 : tensor<\*xf64>  toy.return  }  } |

我们有几个toy.struct\_access访问 toy.struct\_constant. 如 第 3 章（FoldConstantReshape）中所述，我们可以toy通过设置hasFolder操作定义上的位并提供\*Op::fold 方法定义来为这些操作添加文件夹。

|  |
| --- |
| /// Fold constants.  OpFoldResult ConstantOp::fold(ArrayRef<Attribute> operands) { return value(); }  /// Fold struct constants.  OpFoldResult StructConstantOp::fold(ArrayRef<Attribute> operands) {  return value();  }  /// Fold simple struct access operations that access into a constant.  OpFoldResult StructAccessOp::fold(ArrayRef<Attribute> operands) {  auto structAttr = operands.front().dyn\_cast\_or\_null<mlir::ArrayAttr>();  if (!structAttr)  return nullptr;  size\_t elementIndex = index().getZExtValue();  return structAttr[elementIndex];  } |

为了确保 MLIR 在折叠我们的 Toy操作时生成正确的常量操作，即ConstantOpforTensorType和StructConstantfor StructType，我们需要为 dialect hook 提供一个覆盖 materializeConstant。这允许通用 MLIR 操作在必要时为Toy方言创建常量。

|  |
| --- |
| mlir::Operation \*ToyDialect::materializeConstant(mlir::OpBuilder &builder,  mlir::Attribute value,  mlir::Type type,  mlir::Location loc) {  if (type.isa<StructType>())  return builder.create<StructConstantOp>(loc, type,  value.cast<mlir::ArrayAttr>());  return builder.create<ConstantOp>(loc, type,  value.cast<mlir::DenseElementsAttr>());  } |

有了这个，我们现在可以生成可以生成到 LLVM 的代码，而无需对我们的流水进行任何更改。

|  |
| --- |
| module {  toy.func @main() {  %0 = toy.constant dense<[[1.000000e+00, 2.000000e+00, 3.000000e+00], [4.000000e+00, 5.000000e+00, 6.000000e+00]]> : tensor<2x3xf64>  %1 = toy.transpose(%0 : tensor<2x3xf64>) to tensor<3x2xf64>  %2 = toy.mul %1, %1 : tensor<3x2xf64>  toy.print %2 : tensor<3x2xf64>  toy.return  }  } |

您可以自己构建toyc-ch7并尝试toyc-ch7 test/Examples/Toy/Ch7/struct-codegen.toy -emit=mlir：可以在 [DefiningAttributesAndTypes](https://mlir.llvm.org/docs/DefiningDialects/AttributesAndTypes/)中找到有关定义自定义类型的更多详细信息。

# toy语言介绍及其AST的定义

## 语言

本教程将用一种我们称之为“toy”的toy语言来演示（命名很难……）。Toy是一种基于张量的语言，允许您定义函数、执行一些数学计算和打印结果。

考虑到我们想使事情简单化，代码元将被限制为秩<=2的张量，Toy中唯一的数据类型是64位浮点类型（C语言中称为“double”）。因此，所有值都是隐式双精度的，值是不可变的（即每个操作都返回一个新分配的值），解除分配是自动管理的。但用冗长的描述就够了；没有什么比通过一个例子来获得更好的理解更好的了：

|  |
| --- |
| def main() {  # Define a variable `a` with shape <2, 3>, initialized with the literal value.  # The shape is inferred from the supplied literal.  var a = [[1, 2, 3], [4, 5, 6]];  # b is identical to a, the literal tensor is implicitly reshaped: defining new  # variables is the way to reshape tensors (element count must match).  var b<2, 3> = [1, 2, 3, 4, 5, 6];  # transpose() and print() are the only builtin, the following will transpose  # a and b and perform an element-wise multiplication before printing the result.  print(transpose(a) \* transpose(b));  } |

类型检查通过类型推断静态执行；该语言只需要类型声明来在需要时指定张量形状。函数是通用的：它们的参数是无等级的（换句话说，我们知道这些是张量，但我们不知道它们的维数）。它们在调用的位置为每个新发现的签名进行实例化。让我们通过添加一个用户定义函数来重新查看前面的示例：

|  |
| --- |
| # User defined generic function that operates on unknown shaped arguments.  def multiply\_transpose(a, b) {  return transpose(a) \* transpose(b);  }  def main() {  # Define a variable `a` with shape <2, 3>, initialized with the literal value.  var a = [[1, 2, 3], [4, 5, 6]];  var b<2, 3> = [1, 2, 3, 4, 5, 6];  # This call will specialize `multiply\_transpose` with <2, 3> for both  # arguments and deduce a return type of <3, 2> in initialization of `c`.  var c = multiply\_transpose(a, b);  # A second call to `multiply\_transpose` with <2, 3> for both arguments will  # reuse the previously specialized and inferred version and return <3, 2>.  var d = multiply\_transpose(b, a);  # A new call with <3, 2> (instead of <2, 3>) for both dimensions will  # trigger another specialization of `multiply\_transpose`.  var e = multiply\_transpose(b, c);  # Finally, calling into `multiply\_transpose` with incompatible shape will  # trigger a shape inference error.  var f = multiply\_transpose(transpose(a), c);  } |

## AST

上述代码中的AST相当简单；这里对它进行dump如下：

|  |
| --- |
| Module:  Function  Proto 'multiply\_transpose' @test/Examples/Toy/Ch1/ast.toy:4:1'  Params: [a, b]  Block {  Return  BinOp: \* @test/Examples/Toy/Ch1/ast.toy:5:25  Call 'transpose' [ @test/Examples/Toy/Ch1/ast.toy:5:10  var: a @test/Examples/Toy/Ch1/ast.toy:5:20  ]  Call 'transpose' [ @test/Examples/Toy/Ch1/ast.toy:5:25  var: b @test/Examples/Toy/Ch1/ast.toy:5:35  ]  } // Block  Function  Proto 'main' @test/Examples/Toy/Ch1/ast.toy:8:1'  Params: []  Block {  VarDecl a<> @test/Examples/Toy/Ch1/ast.toy:11:3  Literal: <2, 3>[ <3>[ 1.000000e+00, 2.000000e+00, 3.000000e+00], <3>[ 4.000000e+00, 5.000000e+00, 6.000000e+00]] @test/Examples/Toy/Ch1/ast.toy:11:11  VarDecl b<2, 3> @test/Examples/Toy/Ch1/ast.toy:15:3  Literal: <6>[ 1.000000e+00, 2.000000e+00, 3.000000e+00, 4.000000e+00, 5.000000e+00, 6.000000e+00] @test/Examples/Toy/Ch1/ast.toy:15:17  VarDecl c<> @test/Examples/Toy/Ch1/ast.toy:19:3  Call 'multiply\_transpose' [ @test/Examples/Toy/Ch1/ast.toy:19:11  var: a @test/Examples/Toy/Ch1/ast.toy:19:30  var: b @test/Examples/Toy/Ch1/ast.toy:19:33  ]  VarDecl d<> @test/Examples/Toy/Ch1/ast.toy:22:3  Call 'multiply\_transpose' [ @test/Examples/Toy/Ch1/ast.toy:22:11  var: b @test/Examples/Toy/Ch1/ast.toy:22:30  var: a @test/Examples/Toy/Ch1/ast.toy:22:33  ]  VarDecl e<> @test/Examples/Toy/Ch1/ast.toy:25:3  Call 'multiply\_transpose' [ @test/Examples/Toy/Ch1/ast.toy:25:11  var: b @test/Examples/Toy/Ch1/ast.toy:25:30  var: c @test/Examples/Toy/Ch1/ast.toy:25:33  ]  VarDecl f<> @test/Examples/Toy/Ch1/ast.toy:28:3  Call 'multiply\_transpose' [ @test/Examples/Toy/Ch1/ast.toy:28:11  Call 'transpose' [ @test/Examples/Toy/Ch1/ast.toy:28:30  var: a @test/Examples/Toy/Ch1/ast.toy:28:40  ]  var: c @test/Examples/Toy/Ch1/ast.toy:28:44  ]  } // Block |

您可以复制这个结果，并在examples/toy/Ch1/目录中玩这个示例；尝试运行path/to/BUILD/bin/toyc-ch1 test/Examples/Toy/Ch1/ast.toy -emit=ast。

lexer的代码相当简单；它都在一个标题中：examples/toy/Ch1/include/toy/Lexer.h。解析器可以在examples/toy/Ch1/include/toy/Parser.h中找到。它是一个递归下降解析器。如果您不熟悉这样的Lexer/Parser，那么它们与《万花筒教程》前两章中详细介绍的LLVM Kaleidoscope equivalent 非常相似。

下一章将演示如何将此AST转换为MLIR。

# 源码分析

## toy1的源码分析

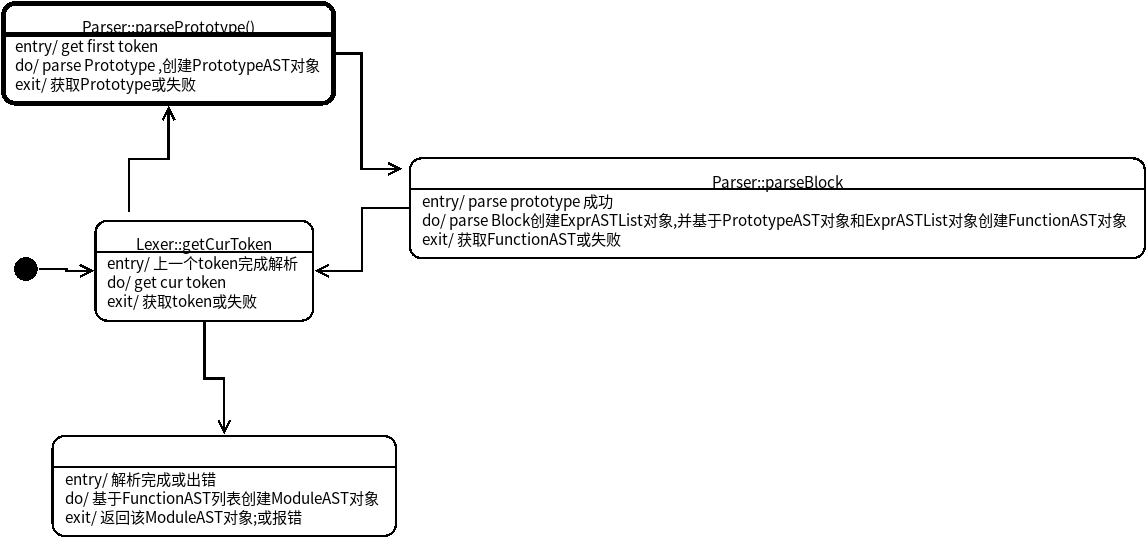
### 简介

make toyc-ch1

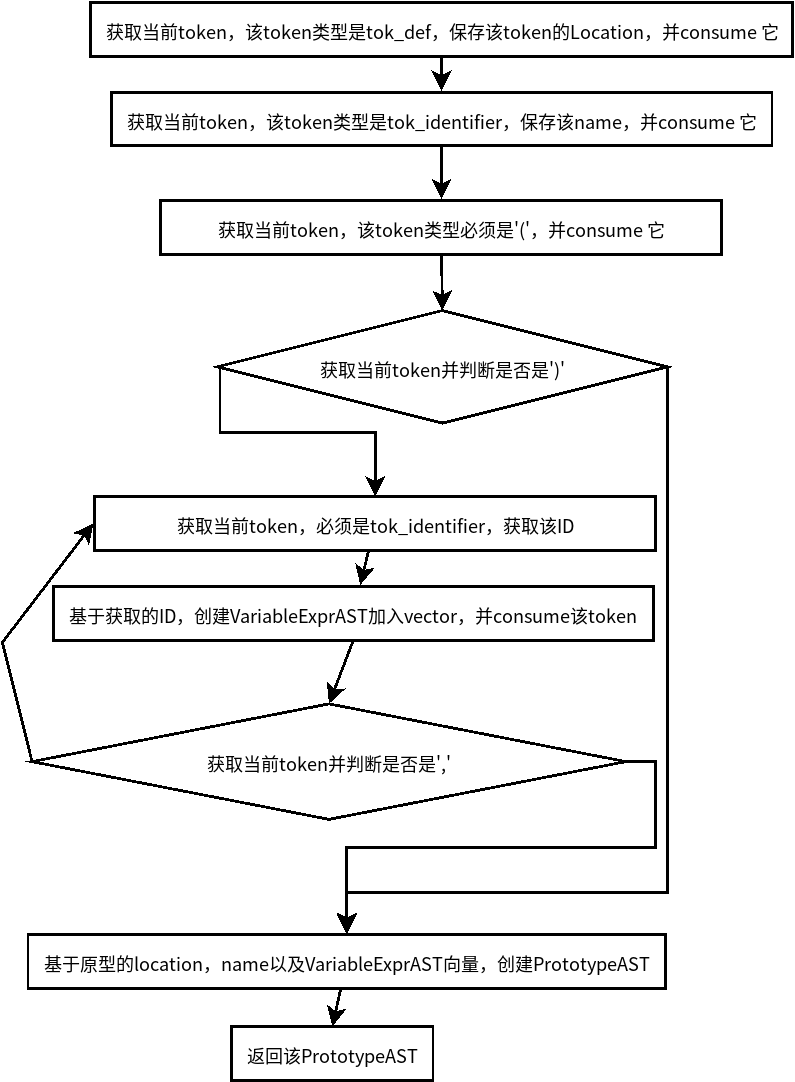
./toyc-ch1 --emit=ast ./toyexample/toy1.ii

### parse的处理流程

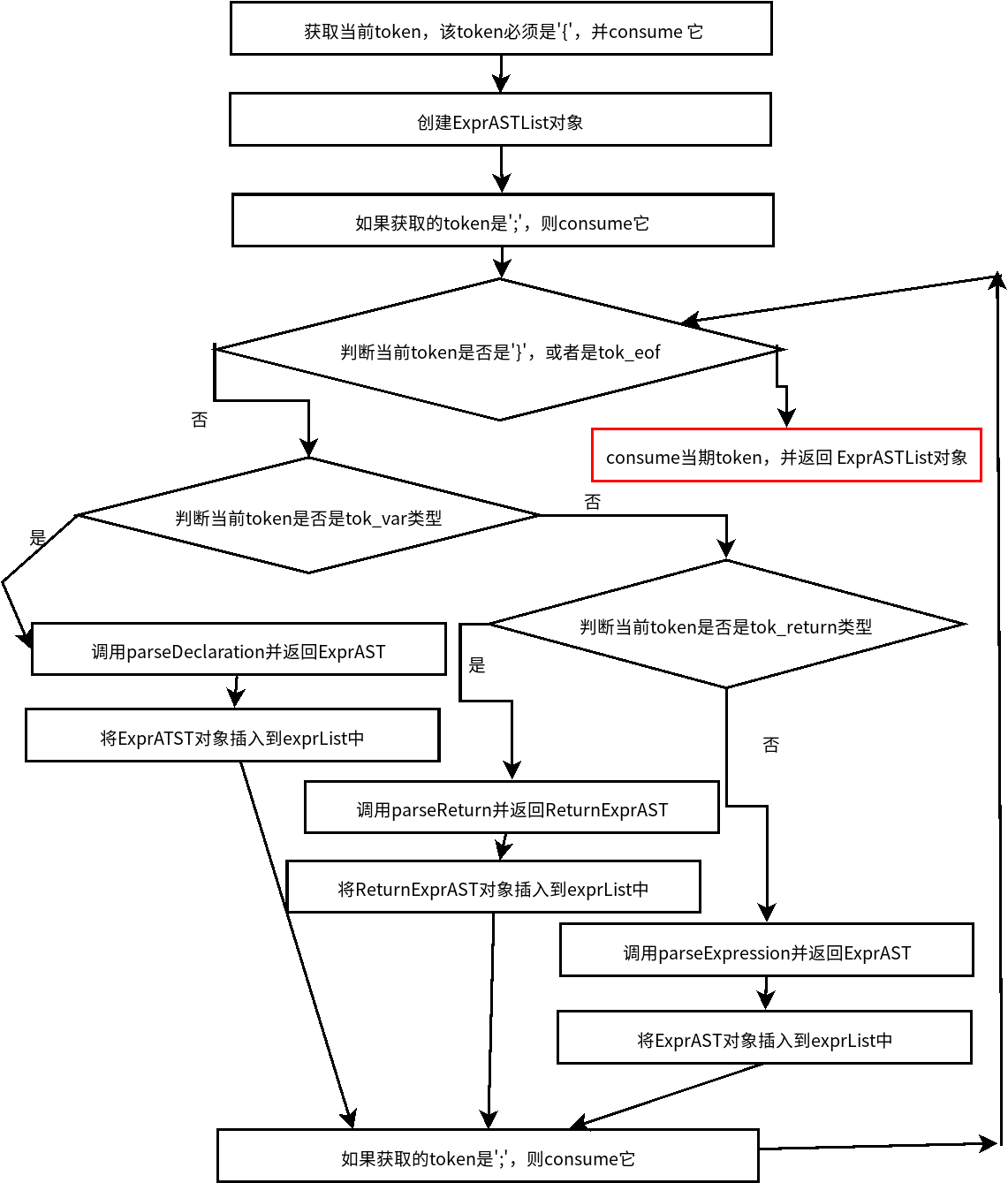
#### parse 基础处理



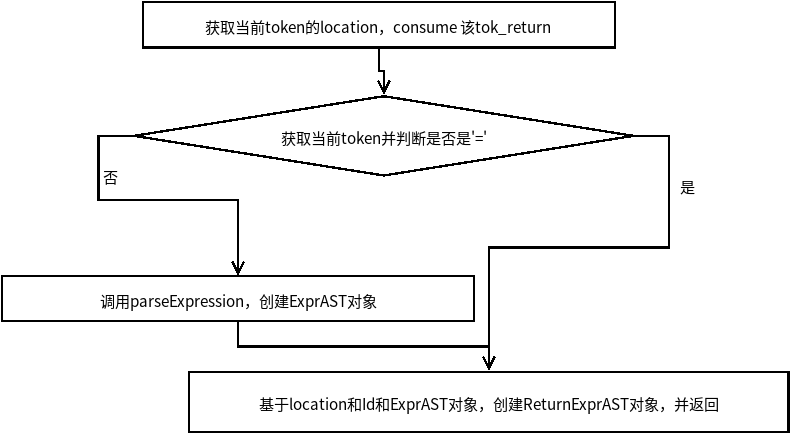
#### parsePrototype处理



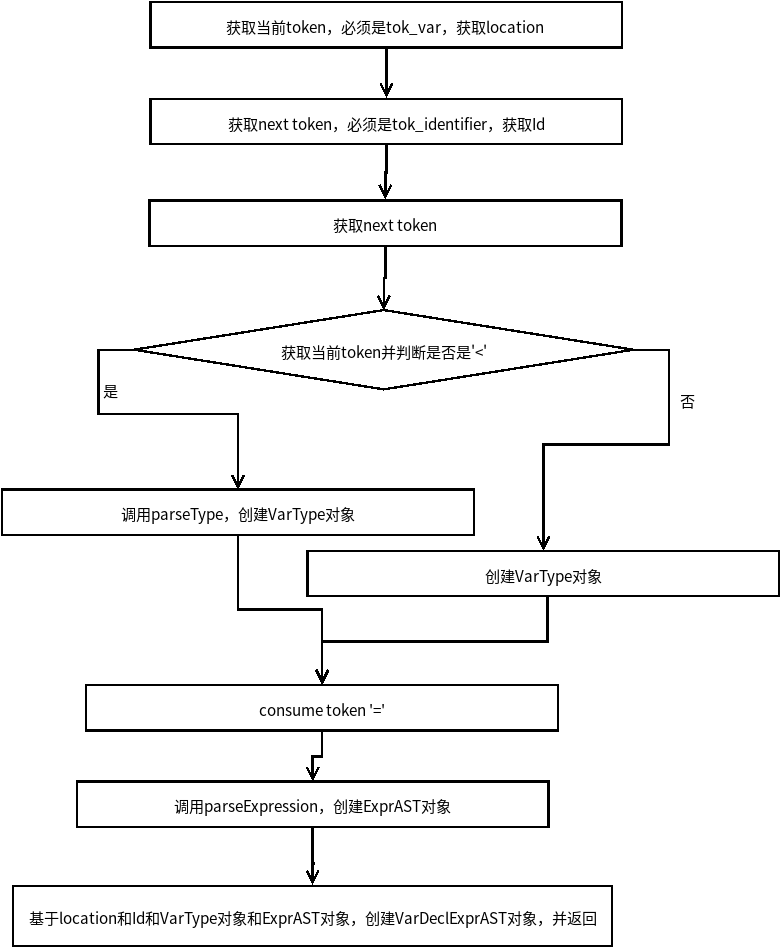
#### parseBlock的处理



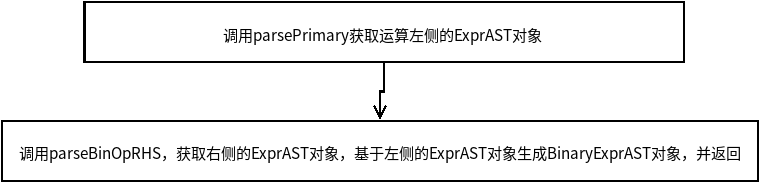
#### parseReturn 处理



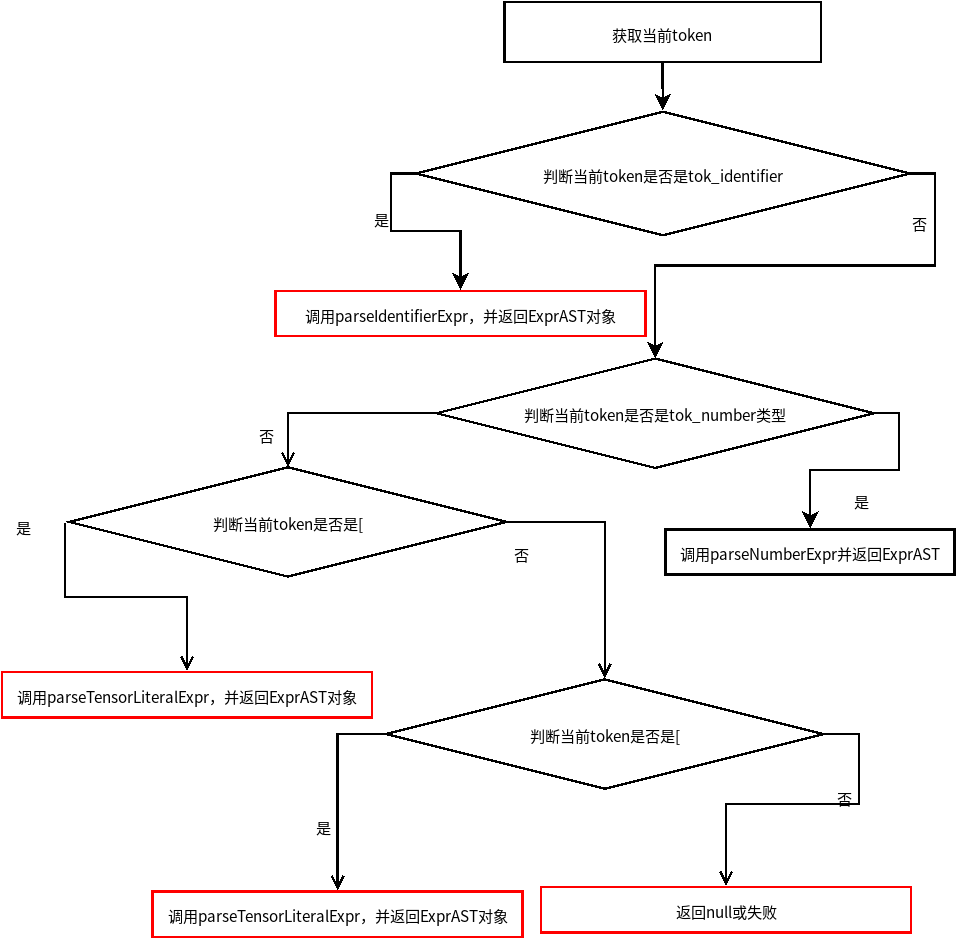
#### parseDeclaration处理



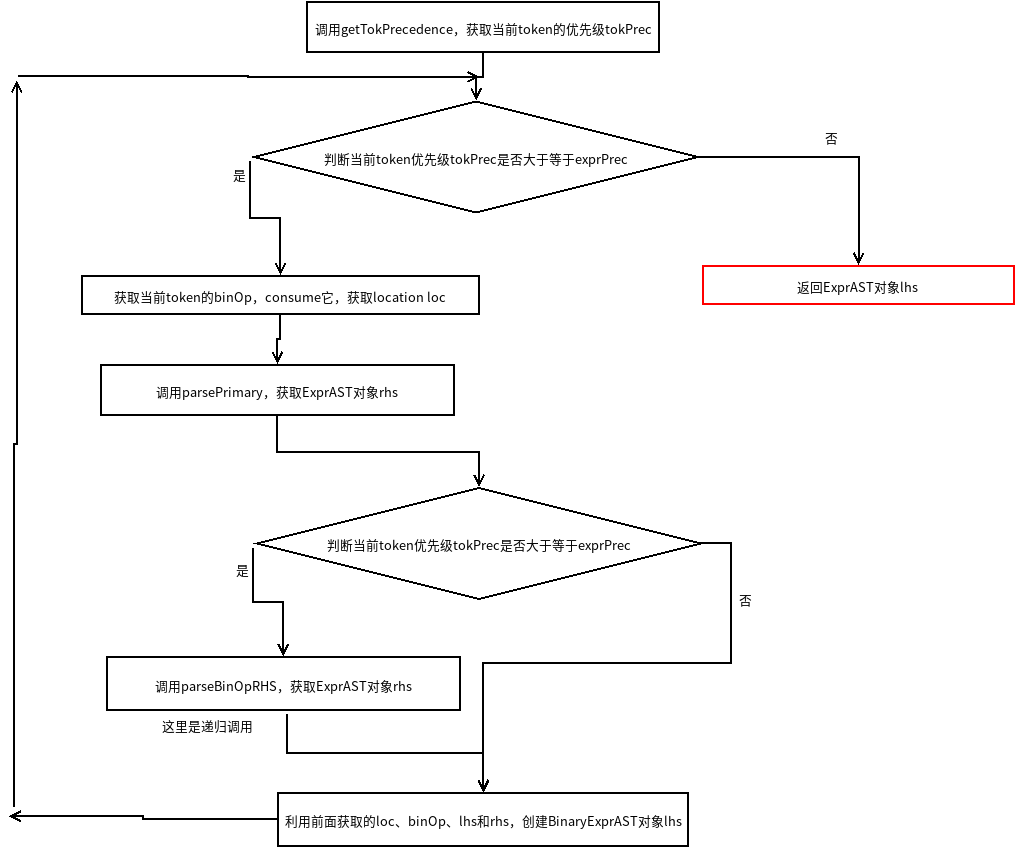
#### parseExpression 处理



#### parsePrimary处理



#### parseBinOpRHS 处理



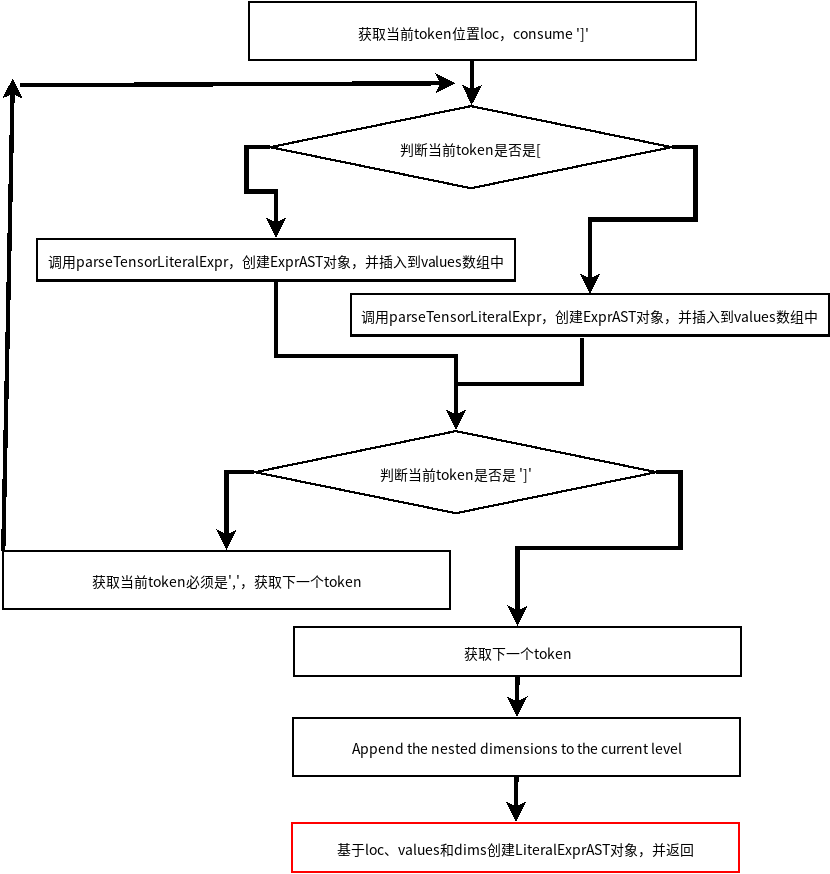
#### getTokPrecedence处理

获取当前token的优先级

#### parseParenExpr处理’()’

1. 当前token是’(‘,consume 它
2. 调用parseExpression，返回ExprAST对象v。
3. 如果获取当前的token不是’)’，则报错。
4. 返回ExprAST对象v。

#### parseTensorLiteralExpr 处理[]



### Lex的处理流程

#### Lexer::getCurToken 、Lexer::getNextToken、Lexer::consume

Lexer::getCurToken:获取当前的token，即curTok属性。

Lexer::getNextToken：调用getTok函数，获取token保存到curTok属性。

Lexer::consume：调用getNextToken函数

#### Lexer::getId、Lexer::getValue

Lexer::getId：返回identifierStr属性。

Lexer::getValue： 返回numVal属性。

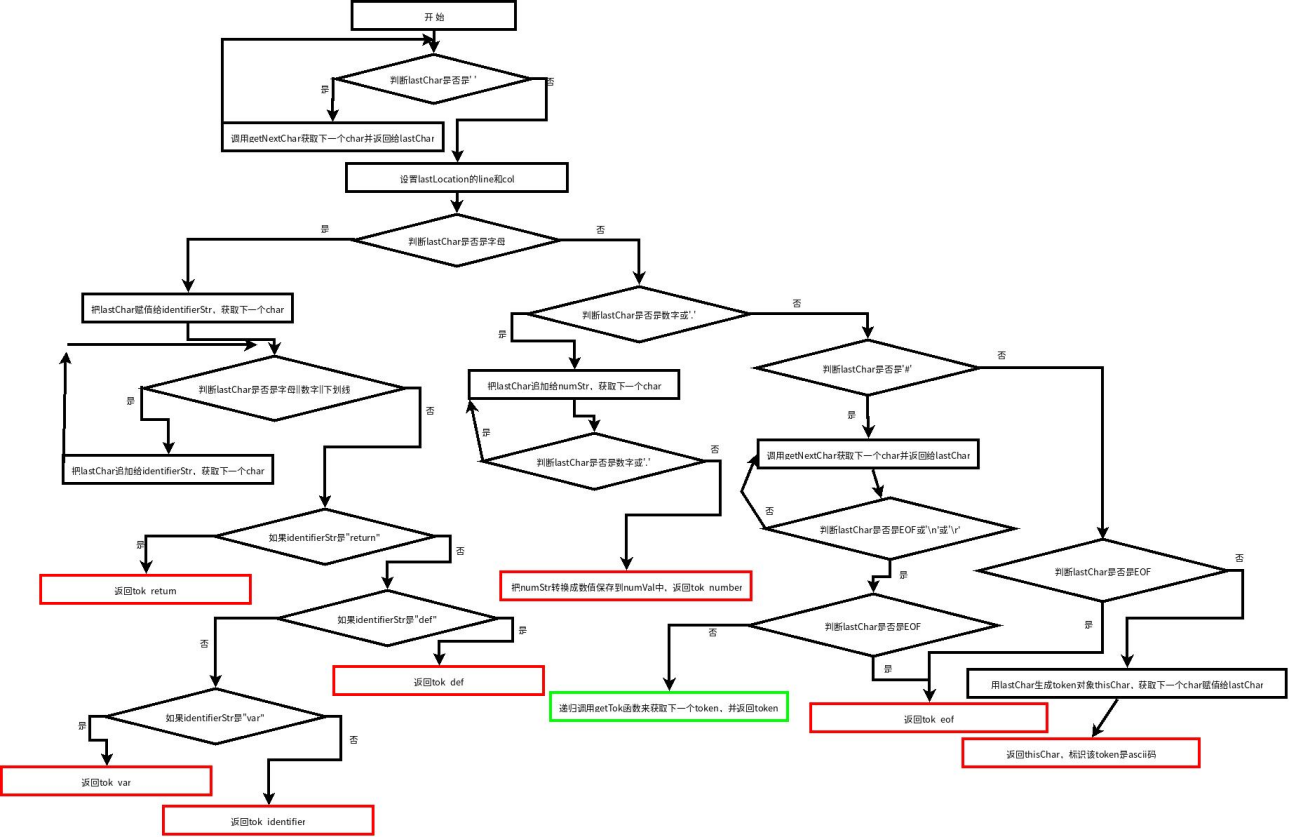
#### Lexer::getLastLocation、Lexer::getLine、Lexer::getCol

Lexer::getLastLocation:返回lastLocation属性

Lexer::getLine：返回curLineNum属性

Lexer::getCol：返回curCol属性

#### Lexer::getTok 从标准输入中获取下一个token



### AST的处理

#### XXX::classof 类操作

判断class类型是否是当前类型，类似于C++中的RTTI。

#### ExprAST::getKind获取当前AST类型

当前AST的类型有

Expr\_VarDecl,

Expr\_Return,

Expr\_Num,

Expr\_Literal,

Expr\_Var,

Expr\_BinOp,

Expr\_Call,

Expr\_Print,

#### ExprAST::loc 获取当前表达式的起始location

获取当前表达式在原始buf中的起始location，包含文件名、行数和列数。

#### 子类的操作列表

* NumberExprAST::getValue 获取数值
* LiteralExprAST::getValues 获取literal expression的对象

该literal expression对象是llvm::ArrayRef<std::unique\_ptr<ExprAST>>类型。因为会嵌套literal express，因此是个数组。

* LiteralExprAST::dims获取literal expression的维度信息
* VariableExprAST::getName 返回变量名
* VarDeclExprAST::getName返回声明名称
* VarDeclExprAST::getInitVal返回初始化表达式
* VarDeclExprAST::&getType 返回类型
* ReturnExprAST::getExpr返回return的表达式或llvm::None
* BinaryExprAST::getOp 返回操作类型
* BinaryExprAST::getLHS返回左侧表达式
* BinaryExprAST::getRHS 返回右侧表达式
* CallExprAST::getCallee 返回调用者名称
* CallExprAST::getArgs 返回调用参数
* PrintExprAST::getArg 获取打印的参数，一个对象
* PrototypeAST::&loc 获取location
* PrototypeAST::getName 获取名称
* PrototypeAST:: getArgs 获取参数，一个VariableExprAST对象的数组
* FunctionAST::getProto 获取PrototypeAST对象
* FunctionAST::getBody 获取函数体，ExprASTList对象
* ModuleAST::begin 返回FunctionAST向量的第一个对象
* ModuleAST::end返回FunctionAST向量的最后一个对象

## toy2的源码分析

### 简介

# 术语

|  |  |
| --- | --- |
| 术语 | 解释 |
|  |  |
| DAG | 有向无环图(Directed Acyclic Graph，DAG) |
| DRR | [声明性重写规则](https://mlir.llvm.org/docs/DeclarativeRewrites/)(DRR，Declarative Rewrite Rules) |
|  |  |
|  |  |
| ODS | Operation Definition Specification (ODS) |
| SSA | static single-assignment，是一种IR(中间表示代码)，要保证每个变量只被赋值一次。这个能帮助简化编译器的优化算法。 |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

# 参考文档

https://llvm-tutorial-cn.readthedocs.io/en/latest/chapter-2.html

https://github.com/tuoxie007/play\_with\_llvm

https://getting-started-with-llvm-core-libraries-zh-cn.readthedocs.io/zh\_CN/latest/ch01.html

https://llvm-tutorial-cn.readthedocs.io/en/latest/index.html

cmake -S llvm -B build -G 'Unix Makefiles' -DCMAKE\_INSTALL\_PREFIX=../install -DLLVM\_INCLUDE\_EXAMPLES=ON -DLLVM\_BUILD\_EXAMPLES=ON -DLLVM\_ENABLE\_PROJECTS="clang;mlir" -DLLVM\_ENABLE\_RUNTIMES="libcxx;libcxxabi" ../llvm-project/llvm

make toyc-ch1

./toyc-ch1 --emit=ast ./toyexample/toy1.ii