**序言**

实验目标： 完成SPL语言编译器，输入为符合SPL语言规范的文本，输出汇编代码和可执行文件。

实验环境：

1. 操作系统：Ubuntu
2. 构建工具：CMake
3. 编译器：GCC
4. 其他工具：Bison，Flex，LLVM

实验设计：

组员分工：

1. **词法分析**
2. 实验原理

词法分析的目标是将源代码根据正则表达式转换成token序列。SPL的token包括标识符、常数值、运算符、分界符、系统函数、系统过程、系统类型、关键字等。除标识符和常数值外，其他token都是某个特定字符串。标识符可以使用（Lex）正则表达式表示：[a-zA-z][\_a-zA-Z0-9]\*，整数值表示为：[0-9]+，实数值可以表示为：[0-9]+(“.”) [0-9]+（在Lex中，.匹配任意单个字符，需要转义处理），字符值可以为表示为\‘.\’（单引号也需要使用转义字符表示）。关键字、系统函数等字符串虽然满足标识符对应的正则表达式，但是不会视为标识符。

Lex源代码主要是一个由正则表达式和相应代码片段组成的表。Lex会将表转换为对应程序，以读取输入流并且将输入划分为匹配正则表达式的字符串，并且识别出对应字符串后会执行相应程序片段。Lex是通过生成确定性有限自动机来进行表达式的识别，并且代码片段会按照输入流中出现的字符串对应表达式顺序执行。

1. **语法分析**
2. 实验原理

语法分析的目标是将token序列转换为parse树。Parse树通过

Yacc源代码主要是一个由上下文无关文法和相应代码片段组成的表，当对应结构被识别出来时会执行相应代码。Bison通过将LALR（1）文法转换为程序来解析语法。

1. **语义分析**

语义分析

1. **优化考虑**

编译时对要生成的代码进行一些优化可以使生成的代码更加漂亮美观，更可以减少最终生成的可执行文件的执行时间，但进行代码优化本身可能会比较复杂，也会增加编译需要的时间。经过对一些优化方法的考虑，我们最终实现了常数折叠与一个比较弱的常数扩散两个优化。

1. 常数折叠

常数折叠是指，把程序中出现的常数表达式在编译期直接计算出结果，生成代码时使用该结果值而不去计算该表达式，以减少运行时的运算量。比如，“a := 2 \* (1 + 3)”会被优化为“a = 8”。

在我们的实现中，我们使用一个结构 ExValue保存值类型的类型与数值，并用一个布尔变量记录该值是否为一个常数。在需要进行运算的地方，先判断所有参与运算的值是否都为常数，如果是，则计算结果并保存到一个新的 ExValue 中用于后续的生成；否则生成运算相应的中间代码，保存一个只有类型而没有数值的非常数 ExValue。

为了便于生成中间代码，ExValue 中也保存了一些其他信息，如下表所示：

|  |  |
| --- | --- |
| 成员 | 含义 |
| union { int val\_i, double val\_r, bool val\_b, char val\_c } | 常数值 |
| bool is\_const | 是否为常数 |
| bool is\_ref | 是否为引用类型 |
| sem::Type type | 值的类型 |
| llvm::Value \*value | 生成时的 llvm 变量 |
| llvm::Value \*addr | 生成时变量的地址 |

在生成表达式的时候，ExValue 对象可作为综合属性，通过函数返回值来传递。

基于以上描述的方法，我们对所有的二元运算（包括二元算数运算、逻辑运算与比较运算）、一元运算（取非与取负）和做计算的系统函数实现了常数折叠。

1. 常数扩散

常数扩散是指，被赋值为常数的变量在接下来的表达式中被视作常数来生成，比如“a := 2; b := a + 1;”，结合常数折叠可以优化为“a := 2; b := 3”。

我们把定义的每一个变量也保存为一个 ExValue，这样，在赋值语句时，如果右侧表达式的结果为常数，那么也修改这个变量的 is\_const 为真，并记录相应的常数值。

然而，这样的常数扩散并不总是正确的。比如，一个循环语句中存在一句“a := a + 1”，而变量 a 在进入循环前被赋值为了常数，如果我们优化为给 a 赋值一个常数，就相当于这个循环只做了一句，这并不是原来的代码想要的结果；类似的情况还可能出现于因为 goto 语句造成的一条语句被多次执行的情况。可见，我们不能对可能会被多次执行的语句执行这样的常数扩散。

除了多次执行造成的问题，调用的函数中也可能会修改接下来要使用的变量，因为在生成函数调用语句时，我们并不能知道函数内部会对哪些变量做怎样的修改，所以我们只能在遇到函数调用时放弃对一个变量拥有常数值的断言。

因此，我们的策略就是，在一个作用域内，一开始承认变量拥有常数值的断言，即根据is\_const 是否为真来进行优化，在遇到第一个标签或第一个函数调用之后，不再承认变量的 is\_const，也就是说，认为所有变量的 is\_const 都为假；此外，在循环语句的内部也不承认该断言。在这种策略下，有写可以做常数扩散优化的地方我们会不去优化，但一定能保证生成正确的结果。

基于以上描述的方法与策略，我们对四种基本类型的变量实现了一个比较弱的常数扩散。

1. **代码生成**

我们使用 llvm 作为生成代码的工具，通过调用 llvm 库提供的方法生成 llvm 中间代码，然后可通过 llvm 提供的命令行工具将该中间代码转换为汇编代码，最后使用 C 语言编译器生成可执行文件。

1. llvm 中间代码

llvm 中间代码从形式上介于C 语言与汇编语言之间，是一个可读性比较高的代码的中间表示。

llvm 中间代码中的变量可以分为全局变量、局部变量与临时变量，其中全局变量在生成汇编代码时会放在数据段；局部变量则是在栈上分配相应大小的空间；临时变量只是用来表示一个计算结果或函数返回值，以在后续表达式中可以使用这个值，没有真正的存储空间。llvm 中间代码的变量存在类型，全局变量与局部变量真正的类型是变量类型的指针，也就是变量的存储地址。

llvm 中间代码的函数与 C 语言类似，有可以为 void 的返回值，有可以为空的参数列表，每一个参数有类型和可选的名字，不允许函数内部出现函数；在形式上也是用大括号括起函数的内容，用小括号括起参数列表；并且也是使用 main 函数作为程序运行的第一个函数。llvm 中间代码的函数并不提供按引用传参的机制，因此需要我们自己实现；此外，不允许函数内部出现函数也要去我们自己想办法解决 SPL 语言中允许的函数内部定义函数。

每一个函数内部有一个或多个代码块，称为 BasicBlock，BasicBlock 有一个名字用以在跳转指令中指示该块为跳转目标，块内有一条或多条语句，且最后一条语句必须是跳转指令或返回指令等，指明该块运行结束后要去哪一个块继续执行。生成时可以通过 IRBuilder 的 setInsertPoint方法来改变中间代码的插入点，新插入的代码总是在一个块的最末尾。BasicBlock 为我们提供了实现控制、循环与跳转语句的基本。

llvm 库中的一个名为 IRBuilder 的类提供了一些方法，这些方法基本上都与一条 llvm 中间代码相对应。此外，也有一些其他类的方法提供了全局变量定义、函数定义的生成。

1. 作用域

在生成的过程中，我们会需要根据一个标识符去查找它是一个变量、一个常量、一个函数名、一个枚举量或是一个类型名。因此，我们仍然需要语义检查中的类型表，以获得名称相应的类型。变量、常量与枚举量都是值，都可以用 ExValue 来表示，不过因为常数扩散中对变量的特殊处理，还是分别定义了变量表与常量表。函数也依然定义了单独的函数表，存储的内容包括函数名、函数参数、函数上级作用域的信息等。此外，跳转标签也需要一个表来记录标签相应的 BasicBlock。

在这些表中，类型表是直接使用语义检查的表。变量表、常量表、函数表与标签表与语义检查时相似，都是一个从字符串（标签表为整数）到相应存储值的哈希表的 vector。每一层作用域对应一个这样的哈希表，到新的作用域时从 vector 尾部插入一个新的哈希表，离开作用域时从 vector 尾部弹出一个哈希表，查找时从尾部向前依次查找，返回找到的第一个元素。

1. 类型处理

在代码生成中，我们不再需要考虑枚举类型与范围类型，枚举类型的本质是整数，范围类型的本质是其去掉范围约束的类型。因此，在生成中将它们直接视作其本质类型。

从我们自己定义的类型到 llvm 中间代码的类型，llvm 提供了 llvm::Type 类来表示类型，并提供了一些方法来构造基本类型与数组、结构等复合类型。我们对基本类型与 llvm 中间代码类型做了如下的对应：

|  |  |
| --- | --- |
| SPL 语言基本类型 | llvm 中间代码类型 |
| integer | i32（32 位有符号整数） |
| real | double |
| char | i8（8 位有符号整数） |
| boolean | i1（1 位有符号整数） |

其中的 i1 虽然是 1 位整数，但其需要的存储空间还是 1 个字节，只是它只有 0 和 1 两个值而已。

llvm 提供了方法 llvm::ArrayType::get 来得到一种类型的定长数组类型，提供了 llvm::StructType::get 来从一组 llvm::Type 得到相应的结构类型，这些能够满足我们 SPL 语言中数组与结构的需要。

1. 常量定义与使用

llvm 库常提供了若干类型的根据值来生成常量的方法，返回是一个 llvm 中间代码的变量，我们将其保存到 ExValue 的 value 字段，并保存到变量表中。如果是表达时中出现的字面常量，那只要将 ExValue 作为综合属性返回即可。

我们需要的 IRBuilder 的大部分方法会接受llvm 变量作为参数，正好我们在 ExValue 中保存的就是这一值，所以不需特殊的处理。

1. 变量定义与使用

就像在第一节描述的那样，我们需要区分全局变量与局部变量，这一点可以通过变量表或其他表的 vector 的大小的来判断。全局变量需要一个值作为其初始化的值，这里我们都赋初值为相应类型的 0；局部变量则通过 llvm 的 IRBuilder 提供的方法 CreateAlloca 来根据类型定义 llvm 中间代码的变量。二者都会返回变量地址，我们将其保存到 ExValue 的 addr 字段，并保存到变量表中。

因为我们保存的是变量地址而非变量本身，在使用时需要通过 CreateLoad 来从这一地址中取出变量值，再用于 IRBuilder 的其他方法之中。不过，赋值语句的目标需要是一个地址，也就是我们保存在 addr 中的内容。

1. 数组与结构的使用

数组元素与结构成员的地址都是通过基地址加偏移量来获取，基地址就是定义变量时返回的地址，而偏移量，对于数组来说，就是它是数组的从 0 开始的第几个元素，对结构来说，就是它是结构的从 0 开始的第几个成员。IRBuilder 提供了方法 CreateGEP（GEP 是 get element pointer 的缩写）来根据基地址和偏移量获取对应的地址。llvm 中间代码记录了类型，会在生成汇编代码时计算出正确的以字节为单位的偏移量。

1. 基本运算与赋值语句

如之前所说，我们把 ExValue 视作综合属性通过返回值来传递，通过一些以 ExValue 为参数的辅助函数，我们可以生成各种运算的中间代码，我们以加法操作为例，介绍我们的方法。

在我们的语法树中，加法操作是一个节点，它的两个子节点分别表示两个运算元，生成两棵子树的内容后，我们通过返回值去到了两个运算元的 ExValue 变量，调用我们自己写的辅助函数 DoAdd，它先进行常数折叠的优化判断，若能优化，则返回一个常数值为运算结果的常量 ExValue，否则调用 IRBuilder 的 CreateAdd 方法，该方法接受两个 llvm 中间代码变量，返回一个变量表示加法结果，将其保存到 ExValue 的 value 字段中，并返回该 ExValue。其他运算操作，包括二元的算数运算、逻辑运算、比较运算和一元的取反与取负数，都是这样的流程。

而赋值语句，IRBuilder 的 CreateStore 方法用以生成赋值的中间代码，我们取出左侧 ExValue 的 addr 字段作为赋值的目的地址，取出右侧 ExValue 的 value 字段作为赋值的源内容，调用 CreateStore 方法。

考虑到变量的 value 字段为空，获取相应的内容需 CreateLoad 的操作，我们为 ExValue 定义了方法 Value() 以处理这样的情况，同时也对 addr 字段定义了相应的方法 Addr() 以正确地取出地址。

1. 布尔逻辑运算的短路计算

按照上一节的方法，布尔类型的逻辑运算会计算两侧表达式的值，但一些情况下只需计算左侧的值就能决定运算结果，从而省去对右侧表达时的计算，这种方式称作短路，是现在大部分高级语言都有的特性。

考虑支持这一特性，我们先只计算左侧表达式的值，根据左表达式的值进行跳转，如果能确定，为一个辅助变量赋值为结果，否则计算右侧表达式的值并赋值给该辅助变量，最后返回该辅助变量对应的 ExValue。在我们的实现中，这一辅助变量是一个名字为“\_#\_bool\_sc”的全局布尔变量，名字中含有‘#’使之不会与其他变量重名。

1. 标签与跳转语句

我们只需在用到标签的时候生成相应的 BasicBlock，在遇到标签语句时，以该标签对于的 BasicBlock 为后续中间代码的插入点；在遇到 goto 语句时执行跳转，由 IRBuilder 的 CreateBr 方法提供，并建立新 BasicBlock 作为 goto 语句之后内容的插入点。

1. 控制与循环语句
2. 函数定义
3. **测试案例**

测试案例