**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**
2. 符号表

符号表是在编译过程中用于处理变量定义、函数定义等内容的一种数据结构，它把符号与变量、函数等内容的对应关系存储起来，以进行未定义的符号、重复定义的符号等关于符号的检查，同时为语义检查的其他部分提供根据符号查找变量、函数等内容的工具。

在 SPL 语言中，每一层作用域都会有标签、常量、类型、变量与函数的定义，相同作用域内不能有重复的符号，但不同的作用域中允许符号的重复定义，且内层符号会遮蔽外层符号。根据这一规则，我们分别为每一层作用域的常量、类型、变量与函数建立一张哈希表（C++ 中的 unordered\_map），从字符串映射到相应的内容；而标签的表比较特殊，因为标签的内容只有标签本身，所以使用哈希集（C++ 中的 unordered\_set）存储即可。此外，为每一层作用域建立一个字符串的哈希集，表示在这一层被定义的符号。定义了以上各表之后，在外层建立各个表的 vector，当开始新的作用域时在 vector 尾部插入一张表，之后的定义对那张新表进行操作，离开作用域时则从尾部弹出，查找时从尾部开始反向依次查找各表，以此实现内层作用域对外层的遮蔽。

在我们的实现中，关于符号表及语义检查的内容均位于 sem 命名空间中。

* 1. 标识符表

每个作用域的标识符表 IdTable 保存一个字符串的哈希集，存储该作用域下定义过的所有标识符，提供了基本的插入与查询方法。

* 1. 类型表与类型

每个作用域的类型表 TypeTable 保存一张字符串到 Type 的哈希表，Type 是我们自己定义的一个用于存储类型的结构，其成员及其含义如表 3-1 所示。

表3-1 Type 类型成员及其含义

|  |  |
| --- | --- |
| 成员 | 含义 |
| enum type | 类型，可能的取值有INTEGER、REAL、CHAR、BOOLEAN、ENUM、ARRAY、RECORD、SUBRANGE、VOID |
| int ind | 该类型是这一层作用域中出现的第几个数组/枚举/结构/范围 |
| int depth | 类型表的作用域的层数，最外层为 0 |
| bool is\_lval | 是否为左值 |

其中 is\_lval 并不是类型的一部分，只是为了做语义检查服务的。VOID 表示错误类型，或过程的返回值。

数组（ARRAY）、枚举（ENUM）、结构（RECORD）、范围（SUBRANGE）这四个复杂类型有各自的结构体来存储只属于它们的信息，数组的具体信息结构 Array 保存数组的下标类型与元素类型，枚举的具体信息结构 Enum 保存从枚举量（字符串）到整数值的哈希表，结构的具体信息结构 Record 按序保存其成员的名称与类型，范围的具体信息结构 Subrange 保存其上下界与元素类型。在一个类型表中，为这四个结构各开一个 vector 来存放那些存储具体信息的结构，而 Type 中的 ind 字段便是具体信息结构在相应 vector 中的下标。

对于基本类型（整数、浮点数、布尔与字符）与 void，类型表提供了静态方法来返回一个 Type 对象，其 type 字段为相应的值，ind 与 depth 字段为 0；而对与复杂类型，类型表也提供了一些方法来构建这些类型，将具体信息结构存储在相应的 vector 中，并返回相应的 Type 对象。对于代码的“type name = xxx”的语句，提供了方法 NameType 来将这一字符串与 Type 对象的映射存储到哈希表中。因此，哈希表中存储的类型只是那些有名字的类型，一些匿名类型，如“var a: array[0..9] of integer”中的数组，只会出现在这四个 vector 中，而不会出现在哈希表中。

在类型的相等判断中，我们有两个判断方法，一个是定义在 Type 类型上的相等运算符，在 type、ind、depth 均相等时视作类型相等，这种判断方法相当于“别名相等”；此外，有一个辅助函数 IsAlmostSame 用于判断更宽松的相等，一是在这里范围类型会取出元素类型来进行比较，二是数组类型会进行“结构相等”的判断，即只要下标类型和元素类型都相等就视为同一类型。此外，亦有一些其他关于 Type 的辅助函数，将在之后介绍。

* 1. 常量表与常量

每个作用域的常量表 ConstTable 保存一张字符串到 Const 的哈希表，Const 是我们自己定义的一个用于存储常量的结构，其成员及其含义如表 3-2 所示。

表3-2 Const 类型成员及其含义

|  |  |
| --- | --- |
| 成员 | 含义 |
| union { int val\_i, double val\_r, bool val\_b, char val\_c } | 常数值 |
| Type type | 常数类型 |

除了在代码的常量定义部分以外，我们把枚举量也视作常量，是常数值为相应的整数、常数类型为相应枚举类型的常量。

常量表除了基本的插入、查询与取出常数值以外，还提供了方法查找是否存在名为某个符号的枚举量。

* 1. 变量表

每个作用域的变量表 VariableTable 保存一张字符串到 Type 的哈希表，提供了基本的插入、查询与取出的方法。

* 1. 函数表与函数

过程是返回值为 void 的函数，故放在一起。

每个作用域的函数表 FuncTable 保存一张字符串到 Func 的哈希表，提供了基本的插入、查询与取出的方法。Func 是我们自己定义的一个用于存储函数签名的结构，其成员及其含义如表 3-3 所示。

表3-3 Func 类型成员及其含义

|  |  |
| --- | --- |
| 成员 | 含义 |
| Type ret | 返回值类型 |
| vector<Type> args | 各个参数的类型 |
| vector<int> mut\_args | 有哪些是可变参数 |

* 1. 标签表

每个作用域的标签表 LabelTable 保存了三个整数的哈希集，分表是在代码的标签定义部分定义的标签、在语句前标注的标签、goto 语句使用的标签。

在 Pascal 语言中，标签需要在代码的标签定义部分定义才能使用，但我们在实验中发现，其实这个部分的存在没有什么必要，但我们还是把它保留了下来。

在一个作用域中，遇见语句前标注的标签则将其插入到标注标签的哈希集中，遇见 goto 语句则将其插入到使用标签的哈希集中，在一个作用域的末尾，如果所有使用标签都在标注标签中出现过，那么通过语义检查，否则不通过。

此外，与其他四项不同，标签只在当前作用域查找，不允许跨作用域的跳转，因为这对函数的活动记录是一个严重的破坏。

* 1. 符号表

在符号表 SymbolTable 这个类型中，我们存放了之前六张表的 vector，进一步封装了它们提供的方法，并使用异常来报出语义错误。

此外，提供了方法 NewScope 与 EndScope 表示作用域的开始与结束，在新建函数或过程时，会调用 NewScope 并在新的作用域中定义参数变量与返回值变量；在检查完一个作用域后，则需要手动调用 EndScope，在该方法中会去检查要弹出的那张标签表是否通过语义检查。

SymbolTable 提供的关于定义变量、常量、函数、类型和命名类型的方法中，均先在最新的一张 IdTable 检查使用存在重名，结构、枚举、数组、范围、函数等还需要进行进一步的判断以检查语义，如数组的下标类型是否真的可以做下标、结构内部是不是存在重名、函数的参数有没有重名等。

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，如表4-1所示。

表 4-1 ExValue 各个成员含义

|  |  |
| --- | --- |
| 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；并基于这种优化，对 if 语句进行了优化，即如果 if 的判断条件为编译器常量，则只生成会执行的部分的代码。

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。

在我们的实现中，关于代码生成与优化的内容均位于 gen 命名空间中。

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对应的 BasicBlock。

在这些表中，类型表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，各自开一个 vector，进入新的作用域时在 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 sem::Type 到 llvm 中间代码的类型 llvm::Type，llvm 库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：

表 5-1 SPL语言基本类型与llvm中间代码类型对应关系

|  |  |
| --- | --- |
| 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。如果是表达式中出现的字面常量，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 IRBuilder 的 CreateLoad 方法来从这一地址中取出变量值，再用于 IRBuilder 的其他方法。不过，赋值语句的目标需要的是一个地址，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，我们通过返回值取到了两个运算元的 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（整数）或 CreateFAdd 方法（浮点数），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非与取负，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式的计算，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为“\_#\_bool\_sc”的全局布尔变量，名字中含有‘#’使之不会与其他标识符重名。

1. 系统函数与系统过程

生成 SPL 语言中系统函数与系统过程的方法可以分为三类：1）用具有相同含义的运算式来替换；2）使用类型转换；3）调用 C 语言函数。

表 5-2 SPL语言部分系统函数可以替换的表达式

|  |  |
| --- | --- |
| 系统函数 | 替换的表达式 |
| succ(x) | x := x + 1 |
| pred(x) | x := x - 1 |
| sqr(x) | x := x \* x |
| odd(x) | x := x and 1 |

如表5-2中描述的那样，这 4 个函数的生成，其实是生成的右侧的运算式，使用第 7 节描述的方法生成。

chr 与 ord 两个函数的功能其实就是进行类型转换，更具体地说，是有符号整数的扩展与截断，IRBuilder 提供了相应的方法来进行扩展与截断

因为我们最后是通过 C 语言编译器来生成可执行文件，所以我们可以使用 C 语言的函数：我们通过调用 C 语言的 sqrt 函数实现 sqrt 函数，调用 abs 或 fabs 函数实现 abs 函数，调用 printf 函数实现 write 和 writeln 函数，调用 scanf 函数实现 read 函数。

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，以该标签对应的 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

各个控制与循环语句都是利用 BasicBlock 配合 IRBuilder 的 CreateBr 方法（无条件跳转）与 CreateCondBr 方法（有条件跳转）来实现的。下图展示了各个控制与循环语句的 BasicBlock 定义与跳转情况。

手机屏幕截图

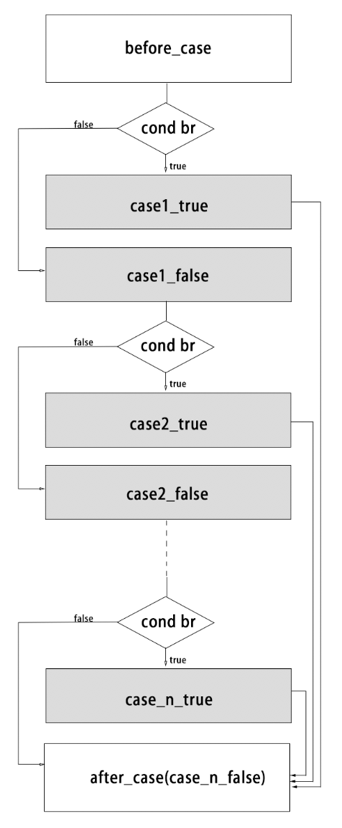
描述已自动生成

图 5-1 if 语句、case 语句的 BasicBlock 跳转关系

手机屏幕截图

描述已自动生成图片包含 游戏机, 截图

描述已自动生成

图 5-2 repeat 语句、while 语句的 BasicBlock 跳转关系

for 循环语句可以视作是一种特殊的 while 循环，为循环变量赋初值的语句为 before\_while 的最后一句，while\_cond 为循环变量与循环终止值之间的比较，循环变量的自增或自减为 while\_content 的最后一句。

1. 函数定义

根据返回值类型与参数类型，使用 llvm 库提供的方法可以建立一个 llvm 中间代码的函数，同时，为每一个参数在函数内作用域建立一个局部变量，然后从 llvm 函数中取出参数值，赋值到相应的局部变量中。如果不是过程，还需为返回值也建立一个局部变量。以上生成了函数最开始的部分。

事实上，建立参数的变量并把参数值赋值这一步有时是不需要的，比如函数内部没有对参数进行过修改，就只需保留这个参数值而不用建立局部变量。不过我们没有对这一点进行判断，依然为每一个参数创建了变量，这是一种更加通用的方式。

函数体的部分与整个程序的生成是类似的，在语法树上就可以看出来。在生成完函数体后，需要生成一条返回语句：如果是过程，通过 CreateRetVoid 生成无返回值的返回语句；否则通过 CreateRet 将返回值变量的值返回。

函数调用的部分，IRBuilder 提供了 CreateCall 方法，该方法需要传进函数的值，也就是 ExValue 中 Value 方法的返回值。

以上描述的是最基本的函数生成的方法，但这样不能实现按引用传参（即实现可变参数）和函数内部定义的函数。

* 1. 按引用传参

要实现按引用传参，根据我们学习的知识，我们知道引用的本质是能够自动解引用的指针，表4-1中 ExValue 的 is\_ref 字段用于表示该值是否为一个引用，引用的 type 字段依然存储的是原类型。

在使用引用类型的变量时，我们也要做到自动解引用。具体地说，引用类型变量所引的变量的地址是引用变量的值，所以引用变量的 Addr 方法返回的是从 addr 中取出来的值而不是 addr 本身，而要获取所引的变量的值，就需要从 Addr 的返回值所代表的地址中再次调用 CreateLoad 取出值。

在生成函数时，如果一个参数是可变参数，则将参数类型设为相应的指针类型；调用时，不使用 Value 方法而是 Addr 方法传入变量的地址。这样，我们就实现了按引用传参。在我们的实现中，如果一个可变参数按引用传参到另一个函数中，因为 ExValue 中如上描述的处理，也是不会出错的。

我们也有考虑过一些其他的实现方法，比如在函数结束时把参数变量的值赋值给一个辅助变量，该变量定义在上一层定义域中，并在调用函数之后，把辅助变量的值赋值给传入函数的变量。但这种实现方式有时会得到错误的结果，比如一个变量既按引用传参传入函数进行修改，其本身也在函数内部被修改。

* 1. 函数内部定义函数

内部函数如果要访问或修改全局变量，那它和普通的函数一样，没有区别，但如果它要访问或修改其上级函数的局部变量，就会出现问题，因为局部函数定义在函数里面，而 llvm 中间代码不允许在函数内部定义函数，因此，在 llvm 中间代码中，内部函数也是一个定义在外侧的函数，因为作用域的关系，它不能访问到上级函数的局部变量。

我们的实现方式是，把所有上层定义的局部变量按引用传参到内部函数，这些参数追加在原来的参数列表的前面，同时保存定义变量时的 ExValue 对象，用以在函数调用时进行参数传递。

手机屏幕截图

描述已自动生成

图 5-3 一段函数内部定义函数的SPL代码片段与相应的我们生成的 llvm 中间代码片段

图5-3左侧是一段 SPL 代码片段，在过程 proc1 内部定义了过程 proc2，它们各有一个整型参数，也各有一个整型局部变量；图 5-3 右侧是生成的 llvm 中间代码的片段，可以看到，proc2 的参数从一个变为了三个，其中原属于 proc1 作用域的整型变量 a 与 g 在 proc2 的参数列表中以指针类型的形式存在，也就是按引用传参。如果 proc2 内部还定义了一个函数，那么 a、g、b、p 会以指针的形式（如上一小节描述的那样，a 和 g 依然会是一级指针）出现在它参数列表的开头。

在这种实现方式中，如果出现了变量重名，因为我们添加的参数是在参数列表的开头，之后定义的同名变量会覆盖我们添加的参数，而根据作用域的规则，确实是我们添加的那些变量被这个内部函数自己的变量给遮蔽了，所以也不会出现问题。

1. **测试案例**

我们在实验过程中测试过的代码整理了一些放在了工程中的 testcases 文件中，这里，我们将对其中的一些进行展示。

1. 课程提供的 3 个测试代码
   1. testcases/spl\_test\_examples/test2.spl

该代码使用递归的方式计算斐波那契数列的第 10 项（以第 1、2 项为 1），主要测试了编译器的函数生成的处理。经我们的编译器编译出的可执行文件给出了 55 的正确答案。

* 1. testcases/spl\_test\_examples/test4.spl

该代码中使用函数递归计算阶乘，把一个全局变量按引用传参到函数中进行修改，该全局变量本身也在函数中进行了修改，造成的结果是，该全局变量最终将等于两倍的 0 到 5 的阶乘和，主要测试了编译器对可变参数的处理。经我们的编译器编译出的可执行文件给出了 308 的正确答案。

* 1. testcases/spl\_test\_examples/test6.spl

该代码使用递归实现辗转相除法计算最大公约数，主要测试了编译器的函数生成的处理。经我们的编译器编译出的可执行文件给出了 27 的正确答案。

1. 测试语义检查
   1. testcases/sem\_check/assign\_stmt.spl

|  |
| --- |
| program assign;  type  student = record  id, age: integer;  gender: char;  end;  var  a, b: integer;  e: real;  arr: array[-5..5] of integer;  r, s: student;  arr2: array[-5..5] of integer;  arr3: array[0..10] of integer;  arr4: array[-5..5] of real;  arr5: array[-5..5] of -5..5;  begin  c := arr['d'];  b := e;  e := b;  e := b.x;  a := e.id;  a := e.gender;  a := r.id;  a := r.gender;  e := r.gender;  a := r.qwq;  r := s;  arr := arr2;  arr := arr3;  arr := arr4;  arr := arr5;  end. |

该代码中，尝试使用不存在的，尝试使用错误的数组下标，尝试访问不存在的结构成员，一些赋值语句也不符合语义，比如把 real 赋值给 integer、把 char 赋值给 integer。

我们的编译器在编译时给出了如下的错误信息：

|  |
| --- |
| assign\_stmt.spl:17: no variable named 'c'  assign\_stmt.spl:17: invalid subscript type  assign\_stmt.spl:18: can't do assignment from rhs type to lhs type  assign\_stmt.spl:20: Accessing member value of a non-record  assign\_stmt.spl:21: Accessing member value of a non-record  assign\_stmt.spl:22: Accessing member value of a non-record  assign\_stmt.spl:24: can't do assignment from rhs type to lhs type  assign\_stmt.spl:25: can't do assignment from rhs type to lhs type  assign\_stmt.spl:26: 'qwq' is not a member of this record  assign\_stmt.spl:29: can't do assignment from rhs type to lhs type  assign\_stmt.spl:30: can't do assignment from rhs type to lhs type |

* 1. testcases/sem\_check/expression.spl

|  |
| --- |
| program expression;  label 6, 7, 8;  type  student = record  id, age: integer;  gender: char;  end;  ENUM = (E\_A, E\_B, E\_C);  var  ea: ENUM;  a, b: integer;  e: real;  arr: array[-5..5] of integer;  r: student;  flag: boolean;  function one(x:integer):integer;  begin  one := 1;  end;  begin  a := 3 \* 4 + 5 mod 6.5;  a := true or false;  a := true and true and (3 < 5) or (6 >= e);  flag := true and true and (3 < 5) or (6 >= e);  flag := true and true and 5;  a := one(a);  7: writeln;  a := one(a, a);  a := one(e);  a := one;  a := abs(a);  flag := abs(flag);  r := succ(r);  r := sqrt(e);  a := 'a';  goto 10;  goto 6;  a := ord('a');  a := ord(chr(ord('a')));  ea := E\_A;  end. |

该代码中，尝试对一个 real 取模，尝试赋值给错误的类型，尝试在布尔逻辑运算中混入整数，尝试以错误的参数调用函数，尝试使用为定义的标签，尝试 goto 一个未标注的标签。

我们的编译器在编译时给出了如下的错误信息：

|  |
| --- |
| expression.spl:21: lhs type and rhs type don't match  expression.spl:21: can't do assignment from rhs type to lhs type  expression.spl:22: can't do assignment from rhs type to lhs type  expression.spl:23: can't do assignment from rhs type to lhs type  expression.spl:25: lhs type and rhs type don't match  expression.spl:28: need 1 parameters, found 2 parameters  expression.spl:29: can't assign the 1st parameter  expression.spl:32: can't assign the 1st parameter  expression.spl:33: can't assign the 1st parameter  expression.spl:34: can't do assignment from rhs type to lhs type  expression.spl:35: can't do assignment from rhs type to lhs type  expression.spl:36: no label called '10' is declared  expression.spl:2: some goto statement goes to an unused label |

* 1. testcases/sem\_check/for\_stmt.spl

|  |
| --- |
| program for\_stmt;  var  i: integer;  j: char;  begin  for i := 1 to 5.5 do begin  writeln;  end;  for i := 1 to 5 do begin  writeln;  end;  for i := 1 to 5 do begin  goto 5;  end;  for i := 'a' to 'z' do begin  writeln;  end;  for i := ord('z') downto ord('a') do begin  writeln;  end;  for j := 'a' to 'z' do begin  writeln;  end;  end. |

该代码中，尝试使用错误的 for 循环上下界，尝试使用为定义的标签。

我们的编译器在编译时给出了如下的错误信息：

|  |
| --- |
| for\_stmt.spl:6: wrong type for second bound of for\_stmt, can't do assignment from rhs type to lhs type  for\_stmt.spl:13: no label called '5' is declared  for\_stmt.spl:15: wrong type for first bound of for\_stmt, can't do assignment from rhs type to lhs type  for\_stmt.spl:15: wrong type for second bound of for\_stmt, can't do assignment from rhs type to lhs type  for\_stmt.spl:3: some goto statement goes to an unused label |

* 1. testcases/sem\_check/proc\_stmt.spl

|  |
| --- |
| program proc\_stmt;  const  real\_c = 5.5;  type  student = record  id, age: integer;  gender: char;  end;  var  i, x: integer;  j: char;  s: student;  b: boolean;  arr: array[-5..5] of student;  procedure multwo(var x:integer);  begin  x := x \* 2;  end;  begin  read(i);  read(x);  read(j);  read('a');  read(s);  read(s.id);  read(arr[5]);  multwo(i);  multwo(j);  multwo(s);  multwo(i, i);  multwo;  multwo(2);  multwo(i + i);  b := b and (i < 5);  end. |

该代码尝试读取一个常量、一个结构，尝试以错误的参数调用 multwo。

我们的编译器在编译时给出了如下的错误信息：

|  |
| --- |
| proc\_stmt.spl:23: must read a variable  proc\_stmt.spl:24: cannot read this type  proc\_stmt.spl:26: cannot read this type  proc\_stmt.spl:28: can't assign the 1st parameter  proc\_stmt.spl:29: can't assign the 1st parameter  proc\_stmt.spl:30: need 1 parameters, found 2 parameters  proc\_stmt.spl:31: need 1 parameters, found 0 parameters  proc\_stmt.spl:32: the 1st parameter is variable and need an lval  proc\_stmt.spl:33: the 1st parameter is variable and need an lval |

1. 生成代码测试
   1. Testcases/gen\_code/array\_for.spl

|  |
| --- |
| program array\_for;  var  a, i: integer;  arr: array[-2..2] of integer;  function get\_two:integer;  begin  get\_two := 2;  end;  begin  a := get\_two;  arr[-2] := a;  for i := -1 to 2 do  begin  arr[i] := arr[i - 1] + 1;  end;  for i := -2 to 2 do  begin  writeln(arr[i]);  end;  end. |

该代码测试了从第一个下标不为 0 的数组中取出元素、for 循环和无参函数。经我们的编译器编译出的可执行文件给出了如下的结果：

|  |
| --- |
| 2  3  4  5  6 |

* 1. testcases/gen\_code/nest\_func\_3.spl

|  |
| --- |
| program nest\_func;  var  oo: integer;  g: real;  procedure proc1(i: integer; var j: integer);  var  a: integer;  b: real;  procedure proc1\_1(r: real);  var  a: real;  function my\_sqrt(i: real):real;  begin  my\_sqrt := sqrt(i);  end;  begin  a := my\_sqrt(r);  b := a;  end;  function proc1\_2(b: integer):real;  var  i: integer;  begin  i := a + b;  proc1\_2 := sqrt(i);  end;  begin  a := i;  proc1\_1(3.3);  g := proc1\_2(4);  end;  begin  proc1(1, oo);  writeln(g);  end. |

该代码测试了函数内部定义函数的一个复杂情况。经我们的编译器编译出的可执行文件给出了 2.236068 的正确结果。

* 1. testcases/gen\_code/var\_func\_3.spl

|  |
| --- |
| program var\_func;  var g: integer;  procedure add2(var a: integer);  begin  a := a + 2;  end;  procedure varproc(var a: integer);  begin  add2(a);  end;  begin  varproc(g);  writeln(g);  end. |

该代码测试了把可变参数按引用传参到另一个函数。经我们的编译器编译出的可执行文件给出了 2 的正确结果。