2022秋compiler–lab3

学号: 201220154　姓名: 王紫萁*

Department of Computer Science, Nanjing University

2022 年 11 月 3 日

1　实验目标

1.1　语义分析

本次实验的目标是实现中间代码生成器。相比实验二语义分析，相同点在于仍然需要利用继承和综合属性做SDT。而不同点在于我们需要根据符号表中记录的信息完成一系列翻译操作。在翻译过程中，有很多地方需要格外注意，因此总结如下。

2　实验内容

2.1　实验环境

- Ubuntu 18.04.6 LTS on WSL2
- GCC version 7.5.0
- GNU Flex version 2.6.4
- GNU Bison version 3.0.4

2.2　完成的任务

基础任务和全部两个扩展任务，并实现了简单且必要的代码优化。

2.3　确定数据结构

中间代码的操作数有以下几种类型：变量（VAR），常量（CONST），地址（ADDR），临时值（TMP），函数（FUN）以及标签（LABEL），对于一个中间代码而言有四种可能的情况出现（实际上是三种大类），单元（single），双元（twin），三元（triple），考虑到条件跳转指令还需要考虑关系运算符，因此在三元下将这种情况单独拎出讨论。

临时值和标签分别具有t_1和$label_i$的格式，常数具有$\#c$的格式，我们为变量和函数型操作数保存他们的名字name，在中间代码中表示为v_name的形式以防止name为临时值名或标签名。

为保证空间利用率，我们设计了`to_string`方法来动态生成每条IR的真实表示形式，在调用时会根据操作数类型和代码类型返回一个新建的三地址代码字符串。

2.4　相对容易的翻译

中间代码主要关注三种变量分别是Stmt，Exp以及Dec。其中Dec也仅关注初始化的变量以及结构体和数组声明，因为前者需要解析Exp来确定变量初值，后者需要为数组和结构体在虚拟机小程序中申请内存。对于其他语句也仅按照他们的最左推导顺序产生综合属性code，最终变量Program的code就是整个程序的中间代码。

Stmt语句会产生的if和while子句，他们的中间代码包括条件跳转和直接跳转。如果直接忠实地按照控制流翻译，为条件的每一种可能都指定一个跳转的目标label并生成跳转，这并不是一件难事。但如此会产生成倍的冗余标签尤其是当条件表达式较复杂时一条语句可能会生成三倍于条件运算符数目的跳转指令。因此在后面关于代码的简单优化部分我们会讨论消除这一冗余。

*E-mail: ziqiwang.wayne@gmail.com

2.5 数组和结构体的翻译

本质上来讲是找寻偏移地址的过程。但在具体实现时可能会有不少阻力，因为以Exp形式呈现的推导式，很难在当前推导确定下一个数组元素的类型，因此也就无法一次性获知每一维的偏移量。同样下一维也看不到前方的基地址。我们以数组为例解答前面的难点。（数组和结构体的混合以及嵌套结构体的多级访问，多维数组只需要处理好翻译array和翻译struct时的交互部分即可）

解决方法是利用好最左推导的性质和综合属性。形如1式(exp[])[]递归解析子式（2式）exp[]的情况中。1式需要得到2式的基地址base和1式自己的元素类型。于是我们为数组产生式Exp -> Exp1[Exp2] 设置综合属性(Operand)address用于存放产生式的基地址在伪代码中存放的位置，属性(Type)elem_t 用于记录当前数组的元素类型。如果Exp1能够直接推导出ID，说明我们推导出了第一维，那么直接在符号表中找到类型信息并将其元素类型放在elem_t返回，同时将address设置成变量的地址[1]。

于是在执行完2式的翻译返回到1式的翻译流程后，我们就拥有了1式的base的地址以及base的类型，那么当前类型就是base.elem_t，偏移地址也可通过在语义分析时得到的数组大小计算得到。

最后需要注意一点的是，在语义分析阶段数组采用Vardec : ID Verdec[INT]—进行自顶向下推导。这种左推导形式不利于我们直接得到当前元素的宽度。例如int a[3][4][5]的语法树的最左下角推导出a[3]，但由于此时我们并不知道a[3]的元素宽度（还未分析到[4][5]）于是便无从得知整个数组的大小。解决方法可以是现记录一遍数组的元素个数，链式存储的array中个数是相反的。上例中会将a的type翻译成 int{3}<-array{4}<-array{5}我们将size倒转，恢复成正确的顺序，即array{3}->array{4}->int{5} 再重新做一次深度搜索，即可正确计算每一维的宽度。

[1]**注意**：如果当前变量是函数参数，那么由于参数已经是退化成指针的变量，因此在后续涉及到地址操作时就不需要取地址了，这一技巧也在结构体或数组直接赋值的翻译中用到

2.6 函数调用的翻译

虚拟机小程序按照入栈顺序解释参数，并按照出栈顺序在函数内部使用参数值。唯一需要注意的地方就是参数类型。如果传入的参数是结构体或数组类型，那么在被调用者内部参数就被解释成地址，此时对参数的使用就不需要取地址操作。判断一个变量是否为形参的流程已在创建符号表是完成，直接检查即可。

2.7 赋值语句的翻译

读者可能会有疑问，如此简单的赋值语句需要浪费笔墨解释吗？首先我们需要明确c–的等号两边可能出现哪些类型的表达式。首先，等号左边必然是基本类型，或者是关于结构体类型和数组类型的推导。基本类型是最简单的一种情况。不太严谨地讲，只要等号左边出现DOT, LB, RB中的一种或多种那这就是一种结构体或数组的推导。一般这种情况都与写内存（Store）语句有关。

需要注意的是，等号左边也可能出现单独一个结构体或数组类型的变量。这种操作的语义是将左侧变量指向右侧变量的空间，此时右侧也必须是同类型的变量（数组类型相同或者结构体类型相同，如let array1[], let array2[], array1 = array2），虽然这是一种糟糕的用法，但也确实存在。此时左侧变量就退化成为一个简单的指针，丢失了自己申请的空间。

在实际翻译赋值语句时，可能会从调用者处传入一个place，这是为了处理连续赋值的情况发生，因此如果 place不为空，我们就需要为其单独生成一条赋值语句。

针对左值的不同类型（left_t），我们可以给出下面的翻译方案：

switch left_t :
  case basic: directly assign
  case array element, structure field :
    store right with the left 's address
  case array, structure:
    make left a pointer and assign the right address to the left

具体如何将左值退化成指针有一个小trick，就是将该变量在符号表中是否为函数参数的属性修改为true。由此一来就能像一个指针一样处理。

2.8 简单有效的优化

2.8.1 表达式的优化

在指导教程中但凡翻译到表达式都会为其新建一个临时值。但是这种做法是低效的，如果表达式能直接推导出`ID`或者`INT`，就可以直接返回，再由调用者决定如何处理这一操作数。例如分析`a = a + b`时用`exp->exp1 + exp2` 推导并翻译会产生如下中间代码：

$$
\begin{aligned}
&t1 = a \\
&t2 = b \\
&t3 = t1 + t2 \\
&a = t3
\end{aligned}
$$

事实上，如果再往前看一步就能直接推导出`a = a + b`从而减少大量冗余代码。

2.9 条件语句优化

这里我们参考了教材[2]上的技术。即将`if`和`while`中的条件语句用`fall`机制简化。他们的`true label` 均设置成`fall`，之后修改条件运算符的翻译模式可以省去大量冗余标签。

3 总结与改进

本次实验完成了中间代码翻译并在翻译基础上做了简单的优化。不足之处在于没有对数组和结构体的寻址做优化导致了很多冗余的寻址操作，在今后的实验中将结合数据流分析等技术进一步缩减代码。

参考文献

[1] 《编译原理实践与指导教程》 许畅等

[2] Aho A V, Lam M S, Sethi R, et al. Compilers: principles, techniques, & tools[M]. Pearson Education India, 2007.