**语义分析实验报告**

魏子洪 2022201289

1. **摘要**

在本次实验中，实现了语法与语义分离，自上而下进行语义分析的SysY编译器。本编译器具有较好的架构，理论上可以处理任何合法的输入，并可以在检测到非法输入时给出错误信息。本编译器的优化能力较差，但也基本没有负优化，综合性能与gcc -O0相近，在部分情况下优于gcc -O0。

1. **编译器架构**

本编译器的流程为：使用flex-bison进行词法和文法分析，生成用于分析的语法树（与文法分析直接所得的语法树不同），随后自上而下进行语义分析，将生成的汇编代码和符号表分别存储于节点和分析器中，最后输出到目标文件，如需要可自动调用gcc编译。

因此架构上，基本按照要编译的节点类型来构造函数，如eval\_exp, global\_compile, func\_compile, stmt\_compile等等。这种架构可以将对寄存器和复杂汇编的操作集中到少数函数中，便于阅读和维护。

1. **重点部分实现**
2. **汇编代码存储**

为了便于对代码做出调整，将汇编代码存储于节点中，在输出时以后序遍历的方式输出，如果无法满足需要，则在语法树中插入仅用于存储汇编代码的节点。汇编代码存储于执行者而非受影响者的节点中，这两者有时并不等同，如一个表达式执行时可能需要将其他表达式的结果入栈来腾出寄存器。

1. **寄存器管理**

根据实际需要，寄存器表对上层提供的接口有：分配、释放寄存器，获取指定寄存器，调用函数时用到的全部入栈、全部恢复，以及为空检测和清空。为了简化实现，临时变量的入栈和出栈全部通过push、pop实现，且编译器不会使用在栈上的临时变量，因此寄存器入栈时，会根据time计数器，选择最早进入寄存器的值入栈，保证最早入栈的值最晚被使用。

编译器并没有语句间保留寄存器表的优化，因此在每条语句执行完成后，会检查寄存器表中是否有值并清空寄存器表，这也是编译器是否正常工作的主要检查点之一。

1. **表达式运算**

表达式运算是编译器的核心，各种复合表达式给寄存器管理带来了很大挑战。在本编译器中，采用了计算-调用-释放的方式来计算表达式和访问数组时的地址。计算函数将立即数或存储结果的寄存器编号存于表达式节点中；调用函数将可能已被入栈的结果取出，并完成操作数类型转换（如立即数->寄存器，内存地址->寄存器）；释放函数将寄存器表中对应数据恢复为空闲状态。计算和调用间可以有大量的操作，而调用和释放间的操作则被严格限制，避免二次入栈。

1. **语法树压缩**

在实现时发现，文法虽然可以很好的处理复杂语句，但生成的语法树并不适合直接分析，存在层层展开的参数列表、几乎无限循环的表达式类型等诸多问题。且自上而下分析需要大量内存存储节点信息，节点数量也需要尽量减少。

因为语法树的节点类型和树状结构已经给出了相当多的信息，因此删去了几乎所有的界符节点，并将参数列表改为单层多个子节点的形式，删去了无分析价值的表达式转换节点，并视需要增添了个别节点。这些操作都在建树时直接完成，而不是建好语法树再做调整。

1. **额外功能**
2. **全局动态初始化**

有一种特殊情况，是在全局变量的初始化中调用函数。这是符合文法和语法的。但全局变量并不在任一函数体中，如果不做处理，生成的汇编代码不仅不会被执行到，还可能会破坏掉其它汇编代码的结构。在本编译器中，如果检测到存在全局动态初始化，会将对应代码追加在生成的汇编代码最后，在刚进入main函数时跳转过去执行。

1. **完全支持布尔运算**

所给的文法中，并不支持将布尔运算和正常运算混合，如不支持1+(a>b)，这也导致了文法不支持布尔表达式中使用括号。本代码中修改了相关文法，并增加了bool到int的转化，以完全支持混合运算，但并不支持bool变量。

1. **错误检查**

作为一个大作业，很难有时间处理好所有的谢姐，比如本编译器虽然在scanf中允许使用指针，但不支持对指针进行运算，如scanf(“%d”,&a-4)。因此，本编译器对不符合语法语义的情况，符合但不支持的情况，编译器自身运行的异常均有检查，并会给出报错信息。另外，在语法阶段具有错误恢复机制，但出现错误后不会进入语义阶段；在语义阶段，出现非编译器错误会立刻终止编译。

1. **编译优化**
2. **控制语句连续跳转**

在处理复合逻辑表达式的时候，如果都写成jl ... jmp ...的形式，很可能会出现原地跳转的情况，因为跳转到的目标紧挨着跳转代码。因此加入了参数next，表示如果不跳转，后面是为条件成立时的分支还是不成立时的分支，或者是其他内容。同理也对return时的跳转做了调整，在函数最末尾的return不需要跳转到目标位置。

1. **常量表达式计算**

常量运算会占用大量的时间，尤其是在数组访问等情况的时候，因此对于可以在编译期间计算的常量表达式，不会进行计算。

1. **不会用到的计算结果**

文法允许计算表达式而不使用其结果，因此，对于不保存结果的指令，只会运行其中的函数。

1. **数组初始化为0**

有些时候，会出现int a[1000]={}这样的写法，需要全部初始化为0，对于这种特殊情况，会使用rep stosl指令填充，而不是使用movl逐一赋值。

1. **优化效果**

仅考虑汇编函数体行数，在给出的10个测试点和自己的2个附加功能性测试点中，本编译器在9个测试点上的行数小于gcc -O0的编译结果。本编译器的平均行数为157.8，而gcc -O0的平均函数为172.6。此处gcc的具体配置为MinGW gcc 13.1.0，使用-masm=att -mno-sse -O0编译（[Compiler Explorer (godbolt.org)](https://godbolt.org/)）。

对于代码的运行效率，分别使用N皇后、素数计数、计算斐波那契数列进行测试，其中前两个程序比gcc -O0快约3%，第三个程序比gcc -O0慢约25%，推测原因为本程序在函数调用和控制语句上速度略高于gcc，但在运算语句上不如gcc -O0。此处gcc的具体配置为gcc version 13.2.0 (x86\_64-win32-seh-rev0, Built by MinGW-Builds project)，使用-O0 -mno-sse -fno-exceptions编译（本地编译）。

综合来看，本编译器的能力与gcc -O0相近。