编译原理 MiniDecaf 编译器实验
Step 1 实验报告

李祥泽
2018011331
lixiangz18@mails.tsinghua.edu.cn

实验内容

1. 开始构造 MiniDecaf 的 EBNF 语法

见 `src/main/antlr/minidecaf/MiniDecaf.g4`.

在这个阶段, 我们可以简单地认为一个**程序** (非终结符, `program`) 中有且只有一个**函数定义** (非终结符, `function`), 后者又可以展开成 "返回值类型 函数名 一对圆括号 左花括号 一条语句 右花括号" 的形式. 其中, 函数名和各种括号都是终结符: 函数名属于**标识符** (`IDENT`) 类型的终结符; 各种括号以字符串字面量的形式给出, 由 ANTLR 自动处理.

语句 (非终结符, `statement`) 目前只有一种, 即**返回语句** (`returnStatement`), 为 "return关键字 表达式 分号" 的形式. **表达式** (`expr`) 目前也只有一种, 即**数字字面量**的终结符 `INTEGER`.

标识符由正则表达式 `[a-zA-Z_][a-zA-Z_0-9]*` 给出, 数字字面量由 `[1-9][0-9]* | '0'` 给出. 后者的表示方法是为了使含有多余前导 0 的字面量 (如 0100) 不能通过编译.

2. 借鉴助教的示例实现, 开始编写含有 Stack Machine IR 的编译器

使用 ANTLR 工具解析定义的语法, 得到一个 Visitor 模式的基类 `MiniDecafBaseVisitor`. 继承该类, 按需求重写各个 `visitX` 方法.

助教提供的 Java 版示例实现采用了单遍编译, 没有 IR. 我在我的实现中加入了使用 Stack Machine 方法的 IR. 具体地说, 实现了 IR 的 `push` 和 `ret` 两个指令, 前者将表达式的值压入栈中, 后者从栈中弹出一个值存入返回值寄存器然后使函数返回. 在各个 `visitX` 方法中, 不是生成汇编代码, 而是生成 IR 代码.

为了从 IR 再进一步生成汇编代码, 加入 `AsmGen` 类. 让 Stack Machine 的栈对应栈内存, 将 `push` 和 `ret` 具体写成相应的 RISC-V 汇编.

`Main` 类表示了编译的大流程, 基本复用了助教的实现 (这里实在没有什么可创新的).

思考题

1. 修改 minilexer 的输入 (`lexer.setInput` 的参数), 使得 lex 报错, 给出一个简短的例子

注意到 minilexer 中有一行 `TokenKind("Error", f".", action="error")`, 即对于不能匹配任何模式的字符报错. 因此只需要输入一个没有在 lexer 中定义的字符即可.

例如, 将 `lexer.setInput` 的参数改为 `"+"`, 即可触发 45 行的 Exception, 报错信息为 `lex error at input position 4`

2. 修改 minilexer 的输入, 使得 lex 不报错但 parse 报错, 给出一个简短的例子

为了实现这一步, 需要用合法的终结符创造一个不匹配任何一个产生式的序列 (其实这是很容易的, 毕竟程序符合语法才比较奇迹).

例如, 输入 `“}{”`, 即可触发 42 行的 Exception, 报错信息为 `syntax error, Int expected but Rbrace found`.

3. 在 RISC-V 中, 哪个寄存器是用来存储函数返回值的?

如果返回值不超过 32 位, 用 `a0` (按编号是 `x10`); 如果超过 32 位还可以同时使用 `a1` (`x11`).

Honor Code

本 Step 中的 `Main.java` 大部, `MainVisitor.java` 大部, 及 `Type.java` 全部借鉴了示例实现.