编译 Project 报告

14307130245 张剑锋

2018 年 1 月 6 日

本项目实现了 MiniJava 语言的编译器前、后端。项目使用 JFlex、Bison 实现了源码到 AST 的转换，并利用 ASM 和 LLVM 等库生成 JVM 字节码和 LLVM 中间表示以及本地代码。

1 原理

1.1 词法分析

词法分析是把输入文件的字符流解析成 token 流的过程。一般情况下，编译器会对所有合法 token 构建一个 DFA 来实现线性时间的解析过程。主流的 Lexer 均使用正则表达式生成 DFA 的实现。本项目选用 JFlex。主要原因是 JFlex 语法接近 flex，使我相当于同时掌握了 flex 的用法。同时，JFlex 可以与 Bison 很好地配合工作。

1.2 句法分析

句法分析以 token 流为输入，输出语法树。不同的生成工具会产生不同的解析算法，比如 ANTLR、JavaCC 会产生 LL 的解析器，而 Bison、Yacc、CUP 产生 LALR 的解析器。本项目使用 Bison。选用 Bison 的主要原因是 Bison 生成 LR 系列的解析器；我以前有使用 LL 解析器的经验，因此这次项目想尝试不同的选择。另一方面的原因是 Bison 在开源软件中非常流行。

2 源码结构

`src/main/java/taiho/minijava` 目录下是主要的源码文件。其中：

- `Lexer.l` 是词法文件，用以生成 `Lexer.java`
- `Parser.y` 是语法文件，用以生成 `Parser.java`；此外还可以生成 LALR 解析器的状态表文件 `Parser.output`
- `Position.java` 是用以描述 token 在源码中的位置的类
- `Analyzer.kt` 是用以进行语义分析的类，用途包括分析类成员、方法参数和局部变量，解析符号和表达式类型
- `Main.kt` 是命令行程序入口

`src/main/java/taiho/minijava/ast` 目录下是 AST 的节点类。
`src/main/java/taiho/minijava/backend` 目录下是后端的实现。其中：

- `astprinter` 是 AST 输出类

- `interpreter` 是解释器
- `bytecode` 是 JVM 字节码生成
- `llvm` 是 LLVM IR 和本地代码生成

其它文件是测试文件。

`sampleout` 目录下放有测试的编译输出，Java 和 LLVM 代码可以在任意平台运行，可执行文件可在 Linux x86_64 下运行。

3 错误修复

运行 `java -jar build/libs/MiniJaba-all.jar -i samples/errors.java` 可以看见示例的含有错误的源文件是如何输出错误信息的:

```
java -jar build/libs/MiniJaba-all.jar -i samples/errors.java
To github.com:swordfeng/MiniJaba.git
   e5082d5..f3e9949  master -> master
[swordfeng@swordfeng-PC ~/Projects/MiniJaba](*´∀`)~$ java -jar build/libs/MiniJaba-all.jar -i samples/errors.java
Error@25:14(490)-25:21(497): syntax error, unexpected INTEGER_LITERAL, expecting IDENTIFIER
Error@25:14(490)-25:13(489): syntax error invalid statement
Error@25:14(490)-25:21(497): syntax error invalid statement
Error@25:28(504)-25:27(503): syntax error invalid statement
Error@25:33(509)-25:34(510): syntax error invalid statement
Error@30:9(602)-30:12(605): syntax error, unexpected int
Error@30:9(602)-30:12(605): syntax error invalid statement
Error@30:9(602)-30:12(605): syntax error invalid statement
Error@30:19(612)-30:20(613): syntax error invalid statement
Error@37:32(810)-37:33(811): lexical error unexpected token /
Error@37:32(810)-37:33(811): syntax error, unexpected UNEXPECTED
Error@37:9(787)-37:28(806): syntax error invalid println statement
Error@37:32(810)-37:33(811): syntax error invalid statement
Error@37:34(812)-37:39(817): syntax error invalid statement
Error@37:39(817)-37:41(819): syntax error invalid statement
Error@38:32(851)-38:37(856): syntax error, unexpected false
Error@38:9(828)-38:28(847): syntax error invalid println statement
Error@38:32(851)-38:37(856): syntax error invalid statement
Error@38:37(856)-38:39(858): syntax error invalid statement
Error@39:31(889)-39:32(890): syntax error, unexpected IDENTIFIER
Error@39:9(867)-39:28(886): syntax error invalid println statement
Error@39:33(891)-39:32(890): syntax error invalid statement
Error@39:33(891)-39:38(896): syntax error invalid statement
Error@39:38(896)-39:40(898): syntax error invalid statement
Error@40:33(931)-40:34(932): syntax error, unexpected ;
Error@40:9(907)-40:28(926): syntax error invalid println statement
Error@40:33(931)-40:34(932): syntax error invalid statement
Error@41:32(964)-41:33(965): syntax error, unexpected ), expecting ;
Error@41:9(941)-41:28(960): syntax error invalid println statement
Error@41:32(964)-41:34(966): syntax error invalid statement
Error@42:33(999)-42:34(1000): syntax error, unexpected }, expecting ;
Error@42:10(976)-42:29(995): syntax error invalid println statement
Error@20:9(395)-20:23(409): type error unknown type YourVisitor
Error@22:9(441)-22:21(453): semantic error redefinition of variable nti
Error@28:9(552)-28:23(566): semantic error assign to undefined variable sadkfj
Error@36:34(771)-36:39(776): type error expected int actual bool
Error@49:27(1205)-49:38(1216): type error expected int actual int[]
[swordfeng@swordfeng-PC ~/Projects/MiniJaba](*´∀`)~$                    2018-01-06 星期六 17:35:36
```

3.1 词法错误

`errors.java` 第 37 行:

```
System.out.println(100 / false);
```

Lexer 遇到无法匹配字符,输出:`Error@37:32(810)-37:33(811): lexical error unexpected token /`

3.2 语法错误

语法错误的处理相比非常复杂。可以看出示例中有大量语法错误，但是错误仅仅报告了非法的语句。在 Bison 中，当发生错误时，解析器会尝试在当前栈上放上一个特殊的 **error** token。如果当前没有规则移

入一个 **error**，解析器会丢弃栈上最后一个符号并再次尝试，直到找到一个能够放入 **error** 的位置。放入 **error** 后，解析器会丢弃当前位置之后的符号直到存在下一个符号对应的规则。这就导致了如果错误处理规则存在疏漏，就容易发生 **error**“吃掉”附近过多 token 的情况。而严密的错误处理规则会导致句法文件中存在比正常规则更多的错误处理规则。因此在本项目中，为了使代码简洁，只处理了在一个语句中发生了错误的情况。一旦错误发生，这个错误所在位置附近的语句会在语法树中被处理成一个非法语句节点，来防止它影响其它语句的解析。

简单的语法错误，如第 30 行：

```
int sadkfj;
```

该错误在于在一个语句后面出现了一个声明。可以看到对应的错误信息：

```
Error@30:9(602)-30:12(605): syntax error, unexpected int
```

3.3 语义错误

本项目处理了几类简单的语义错误。

3.3.1 类型错误

```
System.out.println(100 + false);
```

输出：`Error@36:34(771)-36:39(776): type error expected int actual bool`

3.3.2 符号未定义

```
sadkfj = 2333;
```

输出:`Error@28:9(552)-28:23(566): semantic error assign to undefined variable sadkfj`

3.3.3 符号重定义

```
int nti;
boolean nti;
```

输出:`Error@22:9(441)-22:21(453): semantic error redefinition of variable nti`

语义分析对所有类先进行一次遍历，分析类型和类成员的符号定义和它们的作用域；再对所有方法进行一次遍历，先分析参数和本地变量的定义，再解析所有语句内的符号。

4 额外功能

本项目主要添加的额外功能是一个解释器和两个编译器的后端实现。

4.1 解释器

解释器直接在 JVM 上实现。对于 Int，Bool 和 IntArray 类型直接使用对应的 JVM 类型；对于 Object，解释器实现了一个 Obj 类，用以记录对象的类、父对象和成员变量。所有的成员变量实现在每个 Obj 内的 HashMap 里。

解释器运行过程中也会动态检查变量使用时的类型和声明是否一致，不一致会抛出异常退出。

4.2 字节码编译器

JVM 方法执行的时候，有两个本地内存空间：局部变量表和操作数栈。

数值计算的时候，把两个数值推入操作数栈，执行计算指令，计算指令会弹出两个数值，算得结果推回栈上。JVM 里没有直接对 Bool 类型的操作；所有的 Bool 被编译为 byte，在运算时是当作 int 处理的。

对象方法调用的时候，把对象、方法参数按顺序推到栈上，然后调用对应的方法。由于 Java 中所有方法调用基本都是虚方法调用，因此本项目中所有方法调用都是使用 invokevirtual 指令。

对于被调用方法，所有的参数一开始都被放在局部变量表里。返回时把返回值放在栈上，根据对应类型执行返回指令。

所有对象在创建的时候都必须调用构造方法，构造方法有个特殊的名字<init>。该项目对所有 MiniJava 类实现了默认构造方法，调用时使用 invokespecial 指令。

JVM 运行过程中，变量的类型是动态的：虚拟机没有保证变量表或操作数栈上一个值的类型。不过字节码里对于类成员的类型都有静态定义。如果调用时类型不一致会引发虚拟机异常。

4.3 本地代码编译器

本地代码编译器没有直接生成本地代码指令，而是生成 LLVM IR 再用 LLVM 提用的方法生成本地代码。

MiniJava 和 LLVM 的类型如下对应：

- Bool ⇒ i1
- Int ⇒ i32
- IntArray ⇒ i32*
- Object ⇒ 对应类型的 type 的指针

为了处理动态分发，对于所有的 MiniJava 类，项目生成如下的结构

```
type { base_struct, fields... }
```

对于继承其它类的类来说，base_struct 是基类结构；对于无继承的类来说，base_struct 是如下的虚表指针：

```
{ i8** }*
```

这样就保证了所有对象的开头位置指向它的虚表。

虚表是一个指针数组，每一个成员方法对应一项。子类中的方法若和父类方法有相同名字，则这个方法在虚表中的位置和父类方法位置相同。这样就能实现运行时的动态分发。

数组类型用简单的方式处理：分配数组时比请求大小多一个 int，分配到的内存空间第一个 int 的内容为数组长度，从第二个位置开始才是真正的数组内容。

LLVM IR 中，所有变量的值是不可变的。可变状态需要用 alloc 指令分配内存空间；实际代码中这个空间分配在栈上。堆空间可以用 malloc 指令分配。

本项目中所有的虚表生成的符号为“类名$$”，所有方法生成的符号为“类名$方法名”。方法的第一个参数是 this 指针，其后是原方法的参数依次排列。方法的局部变量都用 alloc 分配在栈上。新数组和新对象都用 memset 初始化为 0；随后数组被写入长度，对象被写入虚表指针。