MiniJava 项目报告

王鹏 15307130185 周霖 15307130124

一. 项目简介

MiniJava 语言是 Java 语言的子集，项目资源中已经对 BNF 做了良好的定义。我们在此 MiniJava 项目中做了这些工作：**词法分析，语法分析，抽象语法树的生成**，以及进一步的**语义分析**和**解释器的实现**。

1. 使用方法

见 READEME.md

2. 工具选择

有很多语法/词法自动分析工具供我们选择，例如：Flex/Bison，Lex/Yacc，Jlex/CUP，JavaCC, ANTLR 等。我们选择自己实现语法和词法的自动分析工具来完成我们抽象语法树的提取。我们自己写工具出于这样几个目的：1.可以更加接近底层，在细节处理上拥有更大的自由度。2.无需受限于其他生成工具的语法。3.能够充分运用课程中所学的内容。

我们全部使用 C++(Standard: 11)来完成整个自动分析系统的构建，没有其他的依赖，因此理论上只要有 C++编译器该系统便能正常运行。(我们的实验环境是 Windows 平台+MSVC，Linux 平台下暂未测试。)

3. 项目结构

我们项目的逻辑部分主要分为三个 class 和一个可执行程序：Lexer, Parser，Interpreter 和 main。Lexer 负责词法，Parser 负责语法和语义分析，Interpreter 负责解释 MiniJava 语言，main 函数负责将三个类连接成一条 pipeline，是整个项目的入口。

4. 项目分工

王鹏负责词法、语法分析和解释器的构造，周霖负责语义分析及其错误修复。

二. 现场报告前完成的工作

1. 词法分析 → Lexer

词法分析的目的是将输入的字符流分解成一个个的 token。我们使用 NFA 来完成这阶段的分析。具体定义的 Token 集合为：

```
ID, INT, BOOLEAN, BIOP, COMMENT, TRUE, FALSE,
CLASS, PUBLIC, STATIC, VOID, MAIN, STRING, EXTENDS, LENGTH,
RETURN, IF, ELSE, WHILE, PRINT, THIS, NEW, COMMA, SEMI, LCUR, RCUR, LSQR, RSQR,
LB, RB, DOT, PLUS, MINUS, EQ, NT, INT_LITERAL,
```

其中大部分 Token 都是 MiniJava 语言的保留字，其它几个正则表达式相关的 Token 我们也可以通过构造 NFA 来解决。我们使用最长匹配原则来解决不必要的冲突。

2. 语法分析 → Parser

首先，我们的 Parser 能够通过给定的规则自动解析文法。我们用如下的方式加入文法规则：(具体的所有文法请阅 Parser.cpp→AddRules())

```
AddRule(STATEMENT, { LCUR, MANY_OR_ZERO, STATEMENT, RCUR }, { LCUR, RCUR }, STATEMENT_S);
AddRule(STATEMENT, { IF, LB, EXPRESSION, RB, STATEMENT, ELSE, STATEMENT}, { IF, LB, RB, ELSE }, STATEMENT_IF_ELSE);
AddRule(STATEMENT, { WHILE, LB, EXPRESSION, RB, STATEMENT }, { WHILE, LB, RB }, STATEMENT_WHILE);
AddRule(STATEMENT, { PRINT, LB, EXPRESSION, RB, SEMI }, { PRINT, LB, RB, SEMI }, STATEMENT_PRINT);
AddRule(STATEMENT, { ID, EQ, EXPRESSION, SEMI }, { EQ, SEMI }, STATEMENT_EQ);
AddRule(STATEMENT, { ID, LSQR, EXPRESSION, RSQR, EQ, EXPRESSION, SEMI }, { LSQR, RSQR, EQ, SEMI }, STATEMENT_ARRAY);
AddRule(EXPRESSION, { EXPRESSION, BIOP, EXPRESSION }, {}, EXPRESSION_BIOP);
AddRule(EXPRESSION, { EXPRESSION, LSQR, EXPRESSION, RSQR }, { LSQR, RSQR }, EXPRESSION_INDEX);
```

AddRule 函数的第一项表示目标符号，第二项是个数组，表示具体规则，第三项表示生成抽象语法树时需要消去的符号，第四项表示生成抽象语法树时的进一步的符号定义。

我们使用 LR(1)分析来完成 Parser 的构建，并且我们不提前计算状态表。状态表在解析的过程中在线生成。

同时，项目说明所规定的 BNF 引入了正则表达式符号*和？。一般情况下我们可以通过进一步拆解，引入新的产生式来描述这两个符号，但是在具体的实现中，我们不这样实现。我们新定义两个 Token: MANY_OR_ZERO 和 ONE_OR_ZERO 来表达*和+，不引入新的产生式而是将**直线式**的产生式转变成 **NFA 的形式**表示来增强其表达能力(我们只需要对 LR(1)算法稍加修改就能使其兼容 NFA 形式)。(具体实现见：Parser.cpp→ManualParser→GetParseTree)

3. 错误分析 → Lexer && Parser

词法错误分析：当 NFA 集合不能进一步接受字符时，Lexer 会报错，具体信息类似于如下：

Lexer 会提示其出错位置和**可能接受的字符**。

语法错误分析:当 LR(1)的 NFA 集合不能接受新的 Token 时,Parser 会报错,具体信息类似于如下:

```
---------------------------- Parser ---------------------------------

Construct Parser: Begin
Add Rule: Done
Construct Parser: Done
Error: at { public static void main main
Can't accept any new token
Your current token is MAIN MAIN
Expect: MAIN LB
Error
```

Parser 会提示其出错位置和**可能接受的 Token**。

4. 抽象语法树生成

LR(1)分析过后，我们得到了一棵语法分析树。然后我们调用 Parser.cpp→ManualParser→FilterParseTree 便得到抽象语法树。具体实现是将不必要的 Token 抽离，然后替换原有 Token 定义。

生成的抽象语法树内容太大，结果可以看 README.md。

三. 现场报告后完成的工作

1.语义分析 && 语义错误分析

具体实现见 Parser.cpp→ManualParser→Analysis

a. 循环 extend 检查：

例如：test.java:

```
class Test {
    public static void main(String[] a){
        System.out.println(1);
    }
}

class Fac extends Caf {}
class Caf extends Fac {}
```

输出：

```
---------------------- Aemantic Analysis ----------------------------

Error: Extends Relation Unvalid.
```

b. class 重复/未定义检查
例如：test.java：

```
class Test {
    public static void main(String[] a){
        System.out.println(1);
    }
}

class Fac {}
class Fac {}
```

输出：

```
---------------------- Aemantic Analysis ----------------------------

Error: Class "Fac" Multiply Defined.

Process finished with exit code 0
```

c. methods 重复/未定义检查
例如：test.java：

```
class Test {
    public static void main(String[] a){
        System.out.println(1);
    }
}

class Fac {
    public int Hello() {
        return 1;
    }
    public int Hello() {
        return 2;
    }
}
```

输出：

```
---------------------- Aemantic Analysis ----------------------------

Error: Method Name "Hello" Multiply Defined.

Process finished with exit code 0
```

d. 变量重复/未定义检查

```
class Fac {
    public int Hello() {
        int x;
        return 1;
    }
    public int Bello() {
        int x;
        return 2;
    }
}
```

```
class Fac {
    public int Hello() {
        int x;
        int x;
        return 1;
    }
    public int Bello() {
        return 2;
    }
}
```

左侧不会报错，而右侧会报错：

```
---------------------- Aemantic Analysis ---------------------------

Error: Var Name "x" Multiply Defined.

Process finished with exit code 0
```

我们采用了命令式风格的符号表，并且实现了撤销池。外部的 scope 和内部的 scope 可以同名。

e. expression 中 operator 的合法性检查：

```
class Fac {
    public int Hello() {
        int x;
        bool y;
        int z;
        x = y + z;
        return x;
    }
}
```

输出：

```
---------------------- Aemantic Analysis ---------------------------

Error: Unvalid Operation.

Process finished with exit code 0
```

2.解释器

我们实现了 Interpreter.cpp，可以真正执行 MinJava 程序。效果如下：

a.官方提供的 Factorial.java，输出：

```
------------------------ Executed Result ---------------------------

MiniJavaOutput >> 3628800
```

b.官方提供的 BinarySearch.java，输出：

```
------------------------ Executed Result --------------------------

MiniJavaOutput >> 20
MiniJavaOutput >> 21
MiniJavaOutput >> 22
MiniJavaOutput >> 23
MiniJavaOutput >> 24
MiniJavaOutput >> 25
MiniJavaOutput >> 26
MiniJavaOutput >> 27
MiniJavaOutput >> 28
MiniJavaOutput >> 29
MiniJavaOutput >> 30
MiniJavaOutput >> 31
MiniJavaOutput >> 32
MiniJavaOutput >> 33
MiniJavaOutput >> 34
MiniJavaOutput >> 35
MiniJavaOutput >> 36
MiniJavaOutput >> 37
MiniJavaOutput >> 38
MiniJavaOutput >> 99999
MiniJavaOutput >> 0
MiniJavaOutput >> 0
MiniJavaOutput >> 1
MiniJavaOutput >> 1
MiniJavaOutput >> 1
MiniJavaOutput >> 1
MiniJavaOutput >> 0
MiniJavaOutput >> 0
MiniJavaOutput >> 999
```

实现细节：
Interpreter 的结构如下：

```
class Interpreter {
public:
  // method
  Interpreter() = default;
  std::string Interprete(ParseTree *tree);
  void GenGlobalClassTable(ParseTree *tree);

  void AddVarDeclaration(ParseTree *tree, SymbolTable &symbols);
  void DelVarDeclaration(ParseTree *tree, SymbolTable &symbols);
  void AddMethodDeclaration(ParseTree *tree, std::map<std::string, Function *> &functions);

  void ExecuteStatement(ParseTree *tree, SymbolTable &symbols);
  BaseClass *EvalExpression(ParseTree *tree, SymbolTable &symbols);

  void PrintData(BaseClass *data);

  // data
  ClassTable class_table_;
};
```

GenGlobalClassTable 用来生成全局的 class 符号表。
Add/DelVarDeclaration 用来更新符号表。
AddMethodDeclaration 用来构造 Function 类(Function 类是一个自定义的、可以被执行的类)
ExecuteStatement 用来在 symbols 环境下执行 statement。
EvalExpression 用来获得表达式的值。

主要流程：生成全局符号表→生成函数定义→在全局符号表环境下执行 MainClass 中的 statement。

诸多细节，涉及到数据类型的判定，递归，符号表的更新，表达式的计算，函数的执行等等，具体实现见 Interpreter.h/Interpreter.cpp。

P.S.该解释器**没有实现继承！**

四. 总结

这次项目属于硬核项目，我们在这次项目中充分运用了编译课程中所学的知识，受益匪浅。