MiniJava 编译器 实验报告

14307130078 张博洋

14307130003 曹景辰

一、实验完成情况

本次实验我们完成了一个完整的能够真正使用的 MiniJava 编译器，它可以直接由 MiniJava 源代码文件生成 EXE 可执行文件。同时，本编译器还带有图形界面语法树显示、友好的错误提示、语义错误恢复等功能。

组员分工、编译方式、运行方式、自动测试的说明详见 README.md 文件。

二、功能演示

```
F:\编译\fdu_compilers_pj\tests>..\src\minijavac\Release\minijavac.exe Factorial.
java
[*] Loading Factorial.java ...
[*] Generating AST ...
[*] Generating type information ...
[*] Generating code ...
 [*] Generating code for main() ...
 [*] Generating code for class Fac ...
  [*] Generating code for Fac::ComputeFac() ...
 [*] Generating virtual function table for class Fac ...
[*] Adding DLL import table ...
[*] Linking ...
 [*] Processing symbols ...
 [*] Calculating address ...
 [*] Relocating ...
 [*] Making EXE ...

Compile successful.

F:\编译\fdu_compilers_pj\tests>out.exe
3628800

F:\编译\fdu_compilers_pj\tests>run_tests.bat
testing BinarySearch.java
  OK
testing BinaryTree.java
  OK
testing BubbleSort.java
  OK
testing Factorial.java
  OK
testing LinearSearch.java
  OK
testing LinkedList.java
  OK
testing myDerivedClassTest.java
  OK
testing QuickSort.java
  OK
testing TreeVisitor.java
  OK
请按任意键继续. . .

F:\编译\fdu_compilers_pj\tests>
```

调用我们的编译器编译 Factorial.java

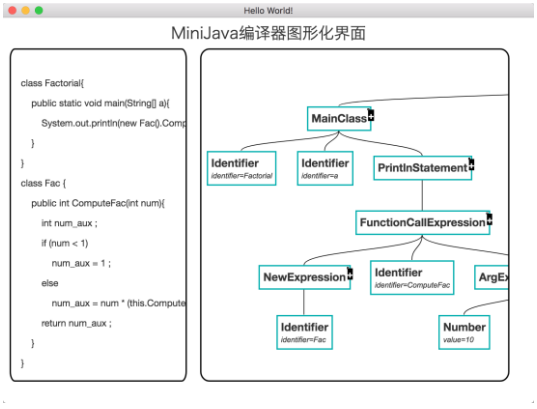
编译成功后，会生成一系列的文件，如：
out.ast.json （JSON 格式的语法树）
out.ast.txt （文本格式的语法树）
out.exe （生成的 EXE 可执行文件）

运行生成的 EXE 文件，可以得到正确的执行结果（10!=3628800）

运行自动测试批处理文件，执行自动测试

显示 OK 则表明我们编译器生成的代码与 JDK 中 JavaC 生成的代码运行结果相同

可以看出，所有样例都通过了测试



图：编译 Factorial.java 以及运行自动测试、图形化界面显示语法树

三、工具选择与原理解释

本次实验我们选择的词法/语法分析工具是 flex 和 bison，它们能够从词法/语法文件生成 C 语言形式的词法/语法分析器。选择它的原因是之前有一门课程也用到了这两个工具，对它们有一定的基础。

flex 是词法分析器的生成工具，它读入“.l”文件，其内含有词法规范的描述信息（正则表达式、动作语句等），生成是一个“.flex.c”文件，即词法分析器的源代码。

bison 是语法分析器的生成工具，它读入“.y”语法文件（文法的定义、语义操作等），生成的是“.tab.c”文件（语法分析器的源代码）和“.tab.h”文件（相关常数的定义等）。

四、源代码结构、核心代码原理、问题与解决思路

1．源代码结构

（1）文件结构

代码文件名	文件功能
minijavac.l	词法描述文件（flex 会生成 minijavac.flex.cpp）
minijavac.y	语法描述文件（bison 会生成 minijavac.tab.cpp/h）
astnode.cpp/h	语法树节点类及其子类
minijavac.cpp/h	代码分析类（前端）
codegen.cpp/h	代码生成类（后端）
jsonvisitor.cpp	负责 JSON 格式语法树的输出的类
printvisitor.cpp	负责文本格式语法树的输出的类
main.cpp	主驱动程序

（2）类结构（部分不重要的类已省略）

类名	功能
ASTNode	语法树节点基类，实现了树节点管理功能，每一种终结符号和非终结符号都有一个对应的子类，以实现相关功能
ASTStatement	语句结点类，实现语句结点的公共功能
ASTIfElseStatement	if … else 语句对应的类
ASTWhileStatement	while 语句对应的类
其它 ASTStatement 的子类已省略	
ASTExpression	表达式结点类，实现表达式结点的公共功能
ASTBinaryExpression	二元运算符对应的类
ASTUnaryExpression	一元运算符对应的类
其它 ASTExpression 的子类已省略	
其它 ASTNode 的子类已省略	
ASTNodeVisitor	对语法树的 Visitor 设计模式的基类
PrintVisitor	负责文本格式语法树的输出的类
JSONVisitor	负责 JSON 格式语法树的输出的类
ClassInfoVisitor	负责从树中获取类信息的类
MethodDeclListVisitor	负责从树中获取类方法信息的类
VarDeclListVisitor	负责从树中获取类变量信息的类
CodeGen	代码生成类，负责为各类型的节点生成代码
MiniJavaC	前端功能综合类
DataItem/DataBuffer/RelocInfo	与机器码、链接器相关的类
TypeInfo/VarDecl/MethodDecl	与类型定义、变量定义、方法定义相关的类

2．核心代码原理

（1）编译器前端

前端主要由 flex 和 bison 实现。具体来说，在词法/语法文件中，对每个非终结符号和终结符号都分配一个相对应的 ASTNode 的子类的实例。若是非终结符号，则把其子节点添加到本节点下；若是终结符号，则把必要的信息（如标识符名、数字的值等）登记在本节点下。这样，主程序中只要调用由 bison 生成的 yyparse() 函数，即可得到源码的语法树了。

```
Statement
  : TOK_LB StatementList TOK_RB
    { $$ = new ASTBlockStatement(@$, { $2 }); }
  | TOK_IF TOK_LP Expression TOK_RP Statement TOK_ELSE Statement
    { $$ = new ASTIfElseStatement(@$, { $3, $5, $7 }); }
  | TOK_WHILE TOK_LP Expression TOK_RP Statement
    { $$ = new ASTWhileStatement(@$, { $3, $5 }); }
  | TOK_PRINTLN TOK_LP Expression TOK_RP TOK_SEMI
    { $$ = new ASTPrintlnStatement(@$, { $3 }); }
  | Identifier TOK_EQUAL Expression TOK_SEMI
    { $$ = new ASTAssignStatement(@$, { $1, $3 }); }
  | Identifier TOK_LS Expression TOK_RS TOK_EQUAL Expression TOK_SEMI
    { $$ = new ASTArrayAssignStatement(@$, { $1, $3, $6 }); }
;
```

图：非终结符号的动作代码的例子

```
"false"         { yylval = new ASTBoolean(yylloc, 0); return TOK_FALSE; }
"true"          { yylval = new ASTBoolean(yylloc, 1); return TOK_TRUE; }
```

图：终结符号的动作代码的例子

语法树生成好后，便可以使用各种类型的 Visitor 对其进行访问了（例如输出 JSON 格式的语法树、代码生成等）。

（2）编译器后端

后端的思路是先遍历一遍语法树，获取类、变量、方法的信息，然后再遍历一遍语法树直接生成代码，最后链接器进行链接、重定位、生成 EXE 的操作。

第一次遍历要做的事情具体分为：（1）获取类名，以及各类之间的继承关系；（2）获取类中成员变量的信息，包括名字、类型，并为它们分配地址；（3）获取类中成员函数的信息，包括名字、返回类型、参数列表、局部变量列表，并为它们在虚函数表中分配位置，此外还要计算参数变量、局部变量的地址）。

```
class Fac:
 class-var:
   (empty)
 method: 00000000 INT ComputeFac()
  arg:
   offset   size     type                 name
   00000000 00000004 INT                  num
   total size 00000004
  local-var:
   offset   size     type                 name
   00000000 00000004 INT                  num_aux
   total size 00000004
```

图：Factorial.java 样例程序中获取到的信息

第二次遍历要做的事情是为各个节点生成代码。对于函数，要生成函数的框架代码（栈帧相关的代码）；对于语句结点，生成必要的控制流代码（如判断、跳转等）；对于表达式，生成基于栈的求值代码。

```
void CodeGen::Visit(ASTWhileStatement *node, int level)
{
    auto beginmarker = DataItem::New();
    auto endmarker = DataItem::New();

    code.AppendItem(beginmarker);
    GenerateCodeForASTNode(node->GetASTExpression());
    PopAndCheckType(node->GetASTExpression()->loc, (TypeInfo { ASTType::VT_BOOLEAN }));
    code.AppendItem(DataItem::New()->AddU8({0x58})->SetComment("POP EAX"));
    code.AppendItem(DataItem::New()->AddU8({0x85, 0xC0})->SetComment("TEST EAX,EAX"));
    code.AppendItem(DataItem::New()->AddU8({0x0F, 0x84})->AddRel32(0x6, RelocInfo::RELOC_REL32, endmarker)->SetComment("JZ end-marker"));

    GenerateCodeForASTNode(node->GetASTStatement());
    code.AppendItem(DataItem::New()->AddU8({0xE9})->AddRel32(0x5, RelocInfo::RELOC_REL32, beginmarker)->SetComment("JMP begin-marker"));

    code.AppendItem(endmarker);
}
```

图：生成 while 语句的代码

```
void CodeGen::Visit(ASTUnaryExpression *node, int level)
{
    GenerateCodeForASTNode(node->GetASTExpression());
    switch (node->op) {
        case TOK_NOT:
            PopAndCheckType(node->GetASTExpression()->loc, TypeInfo { ASTType::VT_BOOLEAN });
            code.AppendItem(DataItem::New()->AddU8({0x83, 0x34, 0xE4, 0x01})->SetComment("XOR [ESP],1"));
            PushType(TypeInfo { ASTType::VT_BOOLEAN });
            break;
        case TOK_LP:
            // nothing to do
            break;
        default: panic();
    };
}
```

图：生成一元运算符表达式的代码

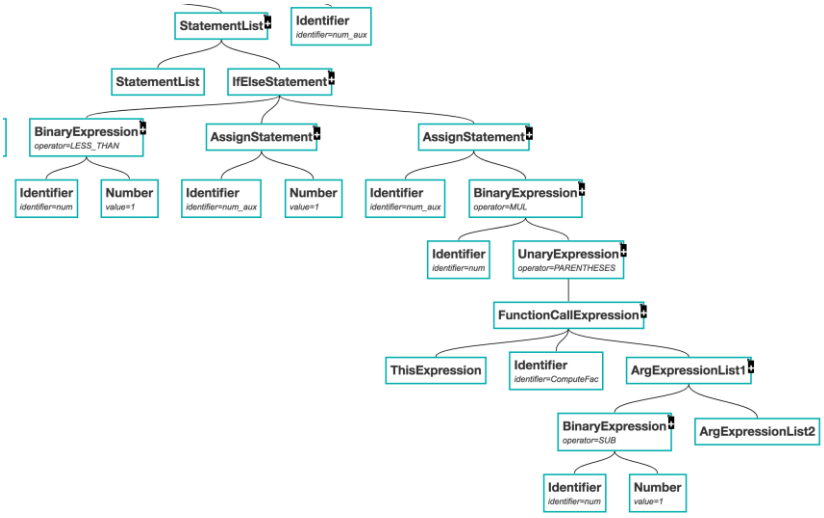
最后调用自己编写的链接器完成链接工作：（1）为所有之前生成的指令分配地址；（2）根据符号表，解决所有未解决的引用（如函数调用等），并进行必要的重定位工作（例如 CALL 指令要求一个相对的偏移量）；（3）将最终代码打包进 EXE 文件。

```
<Fac.ComputeFac>:
 00401041: 55                           PUSH EBP
 00401042: 8B EC                        MOV EBP,ESP
 00401044: 6A 00                        PUSH 0
 00401046: FF B5 0C 00 00 00            load local-var num
 0040104C: 68 01 00 00 00               PUSH ast_number
 00401051: 58                           POP EAX
 00401052: 39 04 E4                     CMP [ESP],EAX
 00401055: 0F 9C C0                     SETL AL
 00401058: 0F B6 C0                     MOVZX EAX,AL
 0040105B: 89 04 E4                     MOV [ESP],EAX
 0040105E: 58                           POP EAX
 0040105F: 85 C0                        TEST EAX,EAX
 00401061: 0F 84 10 00 00 00            JZ else-marker
   reloc :00401077
 00401067: 68 01 00 00 00               PUSH ast_number
 0040106C: 8F 85 FC FF FF FF            store local-var num_aux
 00401072: E9 38 00 00 00               JMP end-marker
   reloc :004010AF
 00401077: FF B5 0C 00 00 00            load local-var num
 0040107D: FF B5 0C 00 00 00            load local-var num
 00401083: 68 01 00 00 00               PUSH ast_number
 00401088: 58                           POP EAX
 00401089: 29 04 E4                     SUB [ESP],EAX
 0040108C: 8B 45 08                     MOV EAX,[EBP+8] (load this)
 0040108F: 50                           PUSH EAX
 00401090: 8B 04 E4                     MOV EAX,[ESP] (eax=this)
 00401093: 8B 00                        MOV EAX,[EAX] (eax=vfptr)
 00401095: FF 90 00 00 00 00            CALL [EAX+vtbloff] (eax=vfptr)
 0040109B: 81 C4 08 00 00 00            ADD ESP,argsize
 004010A1: 50                           PUSH EAX
 004010A2: 58                           POP EAX
 004010A3: F7 2C E4                     IMUL [ESP]
 004010A6: 89 04 E4                     MOV [ESP],EAX
 004010A9: 8F 85 FC FF FF FF            store local-var num_aux
 004010AF: FF B5 FC FF FF FF            load local-var num_aux
 004010B5: 58                           POP EAX
 004010B6: C9                           LEAVE
 004010B7: C3                           RETN
```

图：Factorial.java 样例程序中为 Fac::ComputeFac() 函数生成的代码（已链接完毕）

（3）图形界面

思路是编译器会输出一个 JSON 格式的语法树



编译器会根据输入的 miniJava 代码，生成一个 JSON 文件。JSON 中存储了输入代码的语法树信息。JSON 数据结构如下：

```
src: ‘codes’ // miniJava 源代码
ast: JSON // 包含了抽象语法书节点信息的 JSON
node: { // 单个树节点
  type: // 字符串，说明节点类型
  location: […num] // 数组，标注该节点对应源代码中的位置，由四个数字
                   a,b,c,d 构成，表明该节点代表从 a 行 b 列至 c 行 d 列的代码
  info: // 字符串，节点详细信息
  children : […node] // 子节点数组
}
```

前端读取这个 JSON 文件后，会递归地在内存中构建树的结构。同时，针对每个 node 调用 hover，在现实源代码的同时，每当用户鼠标停留在某个节点上时，源代码中对应部分会高亮出来。

3．问题与解决思路

（1）bison 不支持 (*token*)? 和 (*token*)* 这两种描述

对于第一种情况，可以为它专门设一条产生式，或参照下面的第二种情况进行处理。

对于第二种情况，我们为它专门定义一个 *token*List 的非终结符号，并让它有如下的产生式：

*token*List

：此处留空（允许空表）或设为 *token*（不允许空表）

| *token*List *token*

;

```
MethodDeclarationList
  :
    { $$ = new ASTMethodDeclarationList(@$); }
  | MethodDeclarationList MethodDeclaration
    { $$ = new ASTMethodDeclarationList(@$, { $1, $2 }); }
;
```

图：实际的代码例子

（2）语法中冲突的解决

MiniJava 给出的语法描述较为简单，因此有一些冲突的地方需要解决。对于运算符之间的冲突问题，只要在语法文件中指定运算符的结合性和优先级即可。对于需要向前看更多符号的问题，我们的做法是令 bison 更换分析算法（GLR parser），这样无须改动语法就可以正常工作。

```
%left TOK_EQUAL
%left TOK_LAND
%left TOK_LT
%left TOK_ADD TOK_SUB
%left TOK_MUL
%right TOK_NOT
%left TOK_DOT
```

图：运算符的结合性和优先级定义

（3）错误位置的获取

flex 和 bison 提供了获取 token 位置信息的方法。为了获取错误位置的行号和列号，需要自己定义一个结构体 yyltype，其内含有四个成员 first_line、first_column、last_line、last_column，它们分别代表起始行号、起始列号、终止行号、终止列号。此外，在词法文件中还要手动操作 yylloc 变量的值才能让这个功能正确工作。

```
#define YY_USER_ACTION { \
        yylloc.first_line = yylineno; \
        yylloc.last_line = yylineno; \
        yylloc.first_column = yycolumn; \
        yylloc.last_column = yycolumn + yyleng - 1; \
        for (int i = 0; i < yyleng; i++) { \
            if (yytext[i] == '\n') { \
                yylineno++; \
                yycolumn = 1; \
            } else { \
                yycolumn++; \
            } \
        } \
}
```

图：词法文件中操作 yylloc 变量的代码

正确配置好这个功能后，在词法文件中就可以用 yylloc 获取到当前符号的位置信息了，在语法文件中也可以用 @$ 自动地获取到当前符号的位置信息了。

五、错误检测与修复

本编译器实现了词法/语法的错误检测（无恢复）、语义错误的错误检测和恢复。可以一定程度上帮助程序员修正错误的代码。

本节内容涉及到的代码均可在 mytest 目录下找到。

1．词法/语法的错误检测

（1）未知的 token

```
class MyMain {
    public static void main(String[] a){
        System.out.println(new MyTest().Run());
    }
}

class MyTest {
    public int Run(){
        int x;
        x = x % 100;
        return x;
    }
}
```

```
[*] Loading lexer-unknown-token.java ...
[*] Generating AST ...
 at [(10,8):(10,8)]
   10 |     x = x % 100;
      |         ^
ERROR : syntax error, unexpected TOK_UNEXPECTED


1 error(s) occured, compile failed.
```

图：lexer-unknown-token.java 的代码，其中用到了 MiniJava 没有的取模运算，编译器会报告遇到了未知的 token

（2）错误的语法

```
class MyMain {
    public static void main(String[] a){
        System.out.println(new MyTest().Run());
    }
}

class MyTest {
    public int Run(){
        int int;
        return 0;
    }
}
```

```
[*] Loading parser-unexpected-token.java ...
[*] Generating AST ...
 at [(9,6):(9,8)]
    9 |      int int;
      |          ^~^
ERROR : syntax error, unexpected TOK_INT, expecting TOK_IDENTIFIER


1 error(s) occured, compile failed.
```

图：parser-unexpected-token.java 的代码，其中有一处语法错误（尝试定义 int int），编译器会将其报告出来

2．语义错误

（1）重复的类、变量、方法

对于重复的类成员变量、重复的局部变量、局部变量与参数重复、重复的类方法、重复的类这五种情况，本编译器均可检测并恢复。恢复的原理是忽略掉第二次重复的定义。

```
class Factorial{
    public static void main(String[] a){
        System.out.println(new Fac().ComputeFac(10));
    }
}

class Fac {
    int x;
    int x; // 重复的类成员变量

    public int ComputeFac(int num){
        int y;
        int y; // 重复的局部变量
        int num; // 局部变量与参数重复
        int num_aux ;
        if (num < 1)
            num_aux = 1 ;
        else
            num_aux = num * (this.ComputeFac(num-1)) ;
        return num_aux ;
    }

    public int ComputeFac(int num){ // 重复的类方法
        return 0;
    }

}

class Fac { // 重复的类
}
```

```
[*] Loading sem-duplicate.java ...
[*] Generating AST ...
[*] Generating type information ...
 at [(9,9):(9,9)]
    9 |      int x; // 重复的类成员变量
      |          ^
ERROR : duplicate variable

 at [(13,6):(13,6)]
   13 |      int y; // 重复的局部变量
      |          ^
ERROR : duplicate variable

 at [(11,16):(11,25)]
   11 |      public int ComputeFac(int num){
      |                 ^^^^^^^^^^
ERROR : num exists in both local-var and method-arg

 at [(23,16):(23,25)]
   23 |      public int ComputeFac(int num){ // 重复的类方法
      |                 ^^^^^^^^^^
ERROR : duplicate method

 at [(29,7):(29,9)]
   29 | class Fac { // 重复的类
      |       ^~^
ERROR : duplicate class
```

图：sem-duplicate.java 的代码，编译器可以同时报告多个错误

（2）类型不匹配

对于类型不匹配（无效的类型转换）的问题，本编译器可以检测并报告出来，并进行错误恢复。错误恢复的原理是，假设它们之前能够正常进行类转换即可。

```
class MyMain {
    public static void main(String[] a){
        System.out.println(new MyTest().Run());
    }
}

class MyTest {
    public int Run(){
        int x;
        x = true; // 将布尔值赋给整型变量
        return x;
    }
}
```

```
[*] Loading sem-type-mismatch.java ...
[*] Generating AST ...
[*] Generating type information ...
[*] Generating code ...
 [*] Generating code for main() ...
 [*] Generating code for class MyTest ...
  [*] Generating code for MyTest::Run() ...
 at [(10,2):(10,2)]
   10 |      x = true; // 将布尔值赋给整型变量
      |      ^
ERROR : type mismatch: BOOLEAN, expected INT
```

图：sem-type-mismatch.java 的代码，编译器检测出了类型不匹配的问题，并给出了原类型和期待的类型

（3）未定义的类、变量、方法

对于未定义的变量、方法，本编译器可以检测并报告出来，并进行错误恢复。错误恢复的原理是，假设这个错误的表达式的类型是“未知类型”然后继续编译即可。

```
class MyMain {
    public static void main(String[] a){
        System.out.println(new MyTest().Run2()); // 调用未定义的方法
    }
}

class MyTest {
    public int Run(){
        int x;
        Haha z;
        x = z.test(); // 引用未定义的类中的方法
        return y; // 引用未定义的变量
    }
}
```

```
[*] Loading sem-undeclared.java ...
[*] Generating AST ...
[*] Generating type information ...
[*] Generating code ...
 [*] Generating code for main() ...
 at [(3,34):(3,37)]
    3 |     System.out.println(new MyTest().Run2()); // 调用未定义的方法
      |                                     ^^^^
ERROR : no such method

 at [(3,21):(3,39)]
    3 |     System.out.println(new MyTest().Run2()); // 调用未定义的方法
      |                        ^^^^^^^^^^^^^^^^^^
ERROR : type mismatch: UNKNOWN, expected INT

 [*] Generating code for class MyTest ...
  [*] Generating code for MyTest::Run() ...
 at [(11,6):(11,6)]
   11 |     x = z.test(); // 引用未定义的类中的方法
      |      ^
ERROR : no such class

 at [(11,2):(11,2)]
   11 |     x = z.test(); // 引用未定义的类中的方法
      |     ^
ERROR : type mismatch: UNKNOWN, expected INT

 at [(12,9):(12,9)]
   12 |     return y; // 引用未定义的变量
      |            ^
ERROR : undeclared identifier y

 at [(12,9):(12,9)]
   12 |     return y; // 引用未定义的变量
      |            ^
ERROR : type mismatch: UNKNOWN, expected INT
```

图：sem-undeclared.java 的代码

（4）函数调用的参数与声明不符

对于函数调用时，给出的参数与函数声明不符的问题，本编译器可以检测并报告出来，并进行错误恢复。错误恢复的原理是，直接跳过此函数调用表达式，并令其类型为“未知类型”。

```
class MyMain {
    public static void main(String[] a){
        System.out.println(new MyTest().Run(100)); // 参数数目不正确
    }
}

class MyTest {
    public int Run(){
        return this.Test(true); // 参数类型不正确
    }
    public int Test(int x){
        return 0;
    }
}
```

```
[*] Loading sem-arg-mismatch.java ...
[*] Generating AST ...
[*] Generating type information ...
[*] Generating code ...
 [*] Generating code for main() ...
 at [(3,38):(3,40)]
    3 |     System.out.println(new MyTest().Run(100)); // 参数数目不正确
      |                                         ^^^
ERROR : arg number mismatch

 at [(3,21):(3,41)]
    3 |     System.out.println(new MyTest().Run(100)); // 参数数目不正确
      |                        ^^^^^^^^^^^^^^^^^^^^^
ERROR : type mismatch: UNKNOWN, expected INT

 [*] Generating code for class MyTest ...
  [*] Generating code for MyTest::Run() ...
 at [(9,19):(9,22)]
    9 |     return this.Test(true); // 参数类型不正确
      |                      ^^^^
ERROR : type mismatch: BOOLEAN, expected INT
```

图：sem-arg-mismatch.java 的代码

（5）对非类变量使用成员函数

若对非类变量使用成员函数进行调用，本编译器可以检测并报告出来，

并进行错误恢复。错误恢复的原理是，直接跳过此函数调用表达式，并令其类型为“未知类型”。

```
class MyMain {
    public static void main(String[] a){
        System.out.println(new MyTest().Run());
    }
}

class MyTest {
    public int Run(){
        int x;
        return x.test(); // 对非类变量使用成员函数
    }
}
```

```
[*] Loading sem-invalid-invoke.java ...
[*] Generating AST ...
[*] Generating type information ...
[*] Generating code ...
 [*] Generating code for main() ...
 [*] Generating code for class MyTest ...
  [*] Generating code for MyTest::Run() ...
 at [(10,9):(10,9)]
   10 |     return x.test(); // 对非类变量使用成员函数
      |            ^
ERROR : not a class

 at [(10,9):(10,16)]
   10 |     return x.test(); // 对非类变量使用成员函数
      |            ^^^^^^^^
ERROR : type mismatch: UNKNOWN, expected INT
```

图：sem-invalid-invoke.java 的代码

（6）继承时函数签名不匹配

若在继承时发现派生类的方法声明与基类方法声明不一致（不支持重载），本编译器可以检测并报告出来，并进行错误恢复。错误恢复的原理是，直接忽略此函数定义。

```
class MyMain {
    public static void main(String[] a){
        System.out.println(new MyDerived().Run());
    }
}

class MyBase {
    public int Run(){
        return 0;
    }
}

class MyDerived extends MyBase {
    public int Run(int x){ // 与基类的函数签名不一致
        return x;
    }
}
```

```
[*] Loading sem-invalid-override.java ...
[*] Generating AST ...
[*] Generating type information ...
 at [(14,16):(14,18)]
   14 |     public int Run(int x){ // 与基类的函数签名不一致
      |                ^^^
ERROR : different method prototype
```

图：sem-invalid-override.java 的代码

（7）我们的编译器还实现了更多的语义错误的检测和恢复（例如在 main 函数中使用 this、试图 new 一个未定义的类等），它们的原理都大同小异，因此这里不再赘述。

六、感想

通过一些强大的外部库的帮助，我们亲手构建了 MiniJava 的编译器，更好地了解了编译器的工作原理。一些曾经教科书中抽象的概念和方法，通过亲自编写规则，了解了编译过程的要点。同时，这些知识点和之前的课程，比如计算机原理结合了起来，让我们对计算机科学的知识有了更全面而有深度的理解。