

编译原理

MiniDecaf 编译器实验报告 -- STAGE 3

2021010706 岳章乔

一、思考题

step 7:

1.

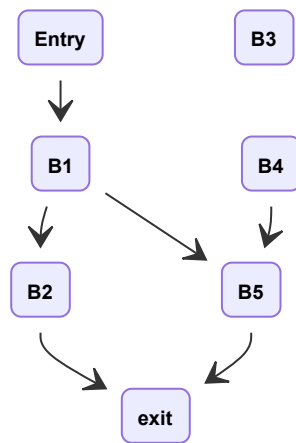
(以本人的 mind 编译器生成的中间代码为例)

```
1  _main:
2      T1 <- 2
3      T0 <- T1
4      T2 <- 3
5      T3 <- (T0 < T2)
6      if (T3 == 0) jump __L1
7      T5 <- 3
8      T4 <- T5
9      return T4
10     return T0
11     jump    __L2
12 __L1:
13 __L2:
14     T6 <- 0
15     return T6
```

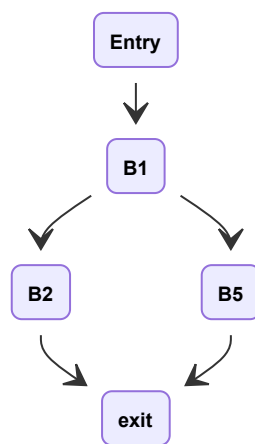
有下列的基本块划分：

```
1  # (B1):
2  _main:
3      T1 <- 2
4      T0 <- T1
5      T2 <- 3
6      T3 <- (T0 < T2)
7      if (T3 == 0) jump __L1
8  # (B2):
9      T5 <- 3
10     T4 <- T5
11     return T4
12 # (B3):
13     return T0
14 # (B4):
15     jump    __L2
16 __L1:
17 __L2:
18 # (B5):
19     T6 <- 0
20     return T6
```

数据流图：



如果把不可达的基本块删除，就是：



其对应

```
1  # (B1):
2  _main:
3      T1 <- 2
4      T0 <- T1
5      T2 <- 3
6      T3 <- (T0 < T2)
7      if (T3 == 0) jump __L1
8  # (B2):
9      T5 <- 3
10     T4 <- T5
11     return T4
12 __L1:
13 # (B5):
14     T6 <- 0
15     return T6
```

step 8:

1.

按汇编指令的行数，理应是第一种会比较好，但按实际上执行过的指令条数，其实两者是等价的。

第一种更接近于原生的 while 循环的翻译，第二种更像是把 while 变换为 do-while 后的中间代码表示：

如果 while 循环有以下形式：

```
1 while(condition)
2     body
```

那么其对应的 do-while 就有以下形式：

```
1 if(condition)
2     do body
3     while(condition)
```

类似的变换其实也可以推广到其他循环形式 -- 比如这次实验的 for 的实现，实际上是在前端的语法分析阶段，在语义行为的部分将之转换为 while 循环。

二、实验内容

2.0. 约定

parser.y

非终结符 `ForStmt` 代表 for 循环语句，`NExpr` 代表可空 (nullable) 表达式。

2.1. 实验需求

实现多层作用域；实现 for 循环和 continue 语句。

2.2. 具体实现

2.3.1. 词法分析

step 8:

(更改1) 补全 `frontend/scanner.l` 的单词表，本次更新针对 for 循环和 continue 语句，分别增加对应的终结符，并将之同步到 `parser.y`。

2.3.2. 语法分析

修改 `frontend/parser.y` 产生式的定义。

step 8:

(更改2) 在 `frontend/parser.y` 增加非终结符 `ForStmt` 的产生式，在语义行为部分转换为 While 循环。

```
1 Stmt      : ForStmt {$$ = $1;}
2 /* Skipped */
3 ForStmt    : FOR LPAREN InitStmt NExpr SEMICOLON NExpr RPAREN Stmt
4             {
5
6                 ast::StmtList* forstmts = new ast::StmtList();
7                 /* 初始化语句 */
8                 forstmts->append($3);
9                 /* 循环体部分：除了即有的 for 循环体语句，也应该在最后加上 step
   的语句 */
10
11                 ast::StmtList* forbody = new ast::StmtList();
12                 forbody->append($8);
13                 forbody->append(new ast::ExprStmt($6, POS(@6)));
```

```

13         ast::CompStmt* forblock = new ast::CompStmt(forbody,
    POS(@8));
14         /* for 的结束条件即为等价的 while 的结束条件 */
15         ast::whileStmt* resultant_while = new ast::whileStmt($4,
    forblock, POS(@1));
16         forstmts->append(resultant_while);
17         $$ = new ast::CompStmt(forstmts, POS(@1));
18     }
19     ;

```

(更改3) 实现 continue 语句，具体类比 break 语句的实现。

```

1 CONTINUE SEMICOLON
2     { $$ = new ast::ContStmt(POS(@1)); }

```

(更改4) 在 `ast/ast_while_stmt.cpp` 实现 continue 语句类的声明和遍历函数：

```

1 ContStmt::ContStmt(Location *l){
2     setBasicInfo(CONTINUE_STMT, 1);
3 }
4
5 void ContStmt::accept(Visitor *v){
6     v->visit(this);
7 }

```

2.3.3. 语义分析

step 7:

(更改5) 编译器第一次遍历语法树，构造符号表，修改 `translation/build_sym.cpp` 对 `CompStmt` 的实现，具体如下。

```

1 Scope *scope = new LocalScope();
2 c->ATTR(scope) = scope;
3 scopes->open(scope);
4
5 // adds the local variables
6 for (auto it = c->stmts->begin(); it != c->stmts->end(); ++it)
7     (*it)->accept(this);
8
9 // closes function scope
10 scopes->close();

```

2.3.4. 中间代码生成

编译器第三次遍历 AST，生成中间三地址代码：

需要修改 `translation/translation.hpp` 和 `translation/translation.cpp`，把 继续循环 节点翻译为三地址码。

(更改6) 标记 `continue` 实际跳转位置，即循环开始标记。

先在 `translation/translation.hpp` 定义该标记：

```

1 tac::Label current_continue_label;

```

(**更改7**) 在 While 语句翻译阶段，记录 continue 的标记；

(**更改8**) 在 `translation/translation.cpp` 实现 continue 语句的翻译（同时在头文件补充声明）：

```
1 void Translation::visit(ast::ContStmt *s) {  
2     tr->genJump(current_continue_label);  
3 }
```

2.3.5. 中间代码优化

略。

2.3.6. 目标代码生成

略