

编译原理 · stage-3 · 实验报告

计01 容逸朗 2020010869

实验内容

实验目标

本阶段需要支持 MiniDecaf 语言的块语句（框架已实现）和循环语句。

具体实现

Continue 语句

首先增加非终结符 `ContStmt` 及对应的语法：

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

然后在语法树中加入对应节点（只需要 `setBasicInfo`），再到 `translation/translation.hpp` 中加入类型为 `tac::Label` 的继续标记 `current_continue_label`。

此时可以在中间码生成时加入对相关标签的处理逻辑即可。

For 循环语句

首先增加非终结符 `ForStmt` 及对应的语法：

```
1 Stmt      : ForStmt    { $$ = $1; }
2           ;
3 ForStmt   : FOR LPAREN ExprStmt OptExpr SEMICOLON OptExpr RPAREN Stmt
4           { $$ = new ast::ForStmt($3, $4, $6, $8, POS(@1)); }
5           | FOR LPAREN VarDecl OptExpr SEMICOLON OptExpr RPAREN Stmt
6           { $$ = new ast::ForStmt($3, $4, $6, $8, POS(@1)); }
7           ;
8 OptExpr   : /* EMPTY */
9           { $$ = nullptr; }
10          | Expr
11          { $$ = $1; }
12          ;
```

此处为了方便语法实现，加入了无实际意义的非终结符 `OptExpr`，同时对语法作了简单的更改（利用 `ExprStmt` 取代 `Expr SEMICOLON`，由于 `init` 只会在循环开始前执行一次，因此这一改动并不会影响程序的正确性）。

根据上面的定义，可以得到如下的语法树节点，需要特别注意的是 `for` 循环有自己的作用域：

```

1  class ForStmt : public Statement {
2      public:
3          ForStmt(Statement *init, Expr *cond, Expr *iter, Statement *body, Location *l);
4          virtual void accept(Visitor *);
5          virtual void dumpTo(std::ostream &);
6
7      public:
8          Statement *init;
9          Expr *cond;
10         Expr *iter;
11         Statement *body;
12         scope::Scope *ATTR(scope);
13 };

```

在符号表构建阶段（SemPass1）时，需要为 `ForStmt` 语句开启新的局部作用域。

```

1  void SemPass1::visit(ast::ForStmt *s) {
2      // opens function scope
3      Scope *scope = new LocalScope();
4      s->ATTR(scope) = scope;
5      scopes->open(scope);
6
7      // adds the local variables
8      if (s->init != NULL) s->init->accept(this);
9      if (s->cond != NULL) s->cond->accept(this);
10     if (s->iter != NULL) s->iter->accept(this);
11
12     s->body->accept(this);
13
14     // closes function scope
15     scopes->close();
16 }

```

类型检查（SemPass2）同理，只需要开启作用域再递归即可。

对于中间码生成的部分，则需要按照 `for` 循环的逻辑编写，同时保留 `label` 方便 `break` 和 `continue` 语句的实验。

```

1  void Translation::visit(ast::ForStmt *s) {
2      Label L1 = tr->getNewLabel();
3      Label L2 = tr->getNewLabel();
4      Label L3 = tr->getNewLabel();
5
6      Label old_break = current_break_label;
7      Label old_continue = current_continue_label;
8
9      current_continue_label = L2;
10     current_break_label = L3;
11
12     if (s->init) s->init->accept(this);

```

```

13     tr→genMarkLabel(L1);
14
15     if (s→cond) {
16         s→cond→accept(this);
17         tr→genJumpOnZero(L3, s→cond→ATTR(val));
18     }
19
20     s→body→accept(this);
21
22     tr→genMarkLabel(L2);
23
24     if (s→iter) s→iter→accept(this);
25     tr→genJump(L1);
26
27     tr→genMarkLabel(L3);
28
29     current_continue_label = old_continue;
30     current_break_label = old_break;
31 }

```

Do While 循环语句

首先增加非终结符 `DoWhileStmt` 及对应的语法：

```

1 Stmt      : DoWhileStmt { $$ = $1; };
2 DoWhileStmt : DO Stmt WHILE LPAREN Expr RPAREN SEMICOLON
3              { $$ = new ast::DoWhileStmt($5, $2, POS(@1)); };

```

整体和 `While` 类似，首先在语法树加入节点，然后在类型检查（`SemPass2`）阶段递归检查。然后生成中间码时先遍历 `loop_body` 再检查 `condition` 即可。

```

1 void Translation::visit(ast::DoWhileStmt *s) {
2     Label L1 = tr→getNewLabel();
3     Label L2 = tr→getNewLabel();
4
5     Label old_break = current_break_label;
6     Label old_continue = current_continue_label;
7
8     current_continue_label = L1;
9     current_break_label = L2;
10
11     tr→genMarkLabel(L1);
12     s→loop_body→accept(this);
13
14     s→condition→accept(this);
15     tr→genJumpOnZero(L2, s→condition→ATTR(val));
16     tr→genJump(L1);
17
18     tr→genMarkLabel(L2);
19 }

```

```
20 |     current_continue_label = old_continue;
21 |     current_break_label = old_break;
22 | }
```

思考题

1. 请画出下面 MiniDecaf 代码的控制流图。

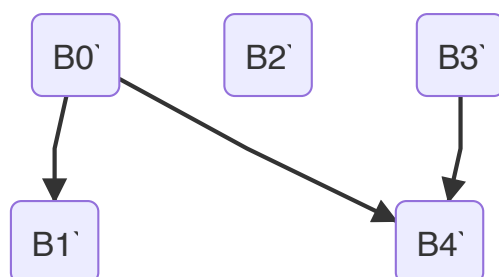
```
1 | int main(){
2 |     int a = 2;
3 |     if (a < 3) {
4 |         {
5 |             int a = 3;
6 |             return a;
7 |         }
8 |         return a;
9 |     }
10| }
```

• 首先将代码翻译为 TAC：

代码	基本块
<code>__main: T1 = 2 T0 = T1 T2 = 3 T3 = (T0 < T2) if (T3 == 0) branch __L1</code>	0
<code> T5 = 3 T4 = T5 return T4</code>	1
<code> return T0</code>	2
<code> branch __L2</code>	3
<code>__L1: __L2: T6 = 0 return T6</code>	4

- 注1：无返回值的函数默认返回值为 0，对应语句在此处为基本块 4。
- 注2：由于讲义和实验指导书在基本块入口定义不同，此处采用实验指导书的方法实现。（即此处保留基本块 2 和 3，若采用讲义的算法则基本块 2 和 3 因不能到达，不应算在基本块入口，在制作控制流图时会被优化）

- 由此可得控制流图如下：



2. 将循环语句翻译成 IR 有许多可行的翻译方法，例如 while 循环可以有以下两种翻译方式：

第一种（即实验指导中的翻译方式）：

1. `label BEGINLOOP_LABEL`：开始新一轮迭代
2. `cond` 的 IR
3. `beqz BREAK_LABEL`：条件不满足就终止循环
4. `body` 的 IR
5. `label CONTINUE_LABEL`：continue 跳到这
6. `br BEGINLOOP_LABEL`：本轮迭代完成
7. `label BREAK_LABEL`：条件不满足，或者 `break` 语句都会跳到这儿

第二种：

1. `cond` 的 IR
2. `beqz BREAK_LABEL`：条件不满足就终止循环
3. `label BEGINLOOP_LABEL`：开始新一轮迭代
4. `body` 的 IR
5. `label CONTINUE_LABEL`：continue 跳到这
6. `cond` 的 IR
7. `bnez BEGINLOOP_LABEL`：本轮迭代完成，条件满足时进行下一次迭代
8. `label BREAK_LABEL`：条件不满足，或者 `break` 语句都会跳到这儿

从执行的指令的条数这个角度（`label` 指令不计算在内，假设循环体至少执行了一次），请评价这两种翻译方式哪一种更好？

- 第二种翻译方式更好，由于把条件跳转语句放到最后的原因，此方式仅用一条指令（7）就可以取代第一种方式的指令（3）和（6），这对于一些需要多次执行循环的情形可以节省不少的指令条数（每循环一次）。
（也许可以把第二种方式的 1 和 2 改为 `br CONTINUE_LABEL` 来缩短翻译后的代码长度？）

参考

实现代码的过程中参考了[实验思路指导与问答墙](#)。