Comparison

SimpleExpression (no Boolean operators yet)

```
if (x == y)        CMP leftItem->reg, leftItem->reg, rightItem->reg
 {                 BNE leftItem->reg, 0, leftItem->fls = 0
  ...              ...
 } else {          BR 0, 0, 0
  ...              ...
 }
```

branch

```
struct item_t {
 ...
 int operator;
 int fls;
 int tru;
};
```

addresses originally unknown!

later for Boolean expressions

```
expressionOperator(struct item_t* leftItem,
                   struct item_t* rightItem,
                   int operator) {
 if ((leftItem->type == INT_TYPE) &&
     (rightItem->type == INT_TYPE)) {
  load(leftItem);

  if ((rightItem->mode != CONST_MODE) ||
      (rightItem->value != 0)) {
   load(rightItem);
   put(CMP, leftItem->reg, leftItem->reg, rightItem->reg);
   releaseRegister(rightItem->reg);
  }
```

optimization

requires extra mode

```
  leftItem->mode = COND_MODE;
  leftItem->type = BOOL_TYPE;
  leftItem->operator = operator;
  leftItem->fls = 0;
  leftItem->tru = 0;
 } else
  error("integer expressions expected");
}
```

CMP, branch are in separate instructions:

z = x == y

if Statement

```
ifStatement() {
 struct item_t* item;
 int fJumpAddress;

 if (symbol == IF) getSymbol(); else error("if expected");
 if (symbol == LPAREN) getSymbol(); else error("missing '('");

 item = malloc(sizeof(struct item_t));
 expression(item);

 if (item->type == BOOL_TYPE) {
  loadBool(item);
  cJump(item);
  fixLink(item->tru);
 } else error("boolean expression expected");

 if (symbol == RPAREN) getSymbol(); else error("missing ')'");

 if (symbol == BEGIN) {
  getSymbol();
  statementSequence();
  if (symbol == END) getSymbol(); else error("missing '}'");
 } else statement();

 if (symbol == ELSE) {
  getSymbol();
  fJumpAddress = fJump();
  fixLink(item->fls);

  if (symbol == BEGIN) {
   getSymbol();
   statementSequence();
   if (symbol == END) getSymbol(); else error("missing '}'");
  } else statement();

  fixUp(fJumpAddress);
 } else fixLink(item->fls);
}
```

later for Boolean expressions

jump here if comparison evaluates to false!

Branching and Fixup

```
cJump(struct item_t *item) {
 put(branch(negate(item->operator)), item->reg, 0, item->fls);
 releaseRegister(item->reg);
 item->fls = PC - 1;
}
```

remember address of branch instruction for later fixup

```
int fJump() {
 put(BR, 0, 0, 0);
 return PC - 1;
}
```

return the address of the instruction we just emitted for later fixup

```
fixUp(int branchAddress) {
 encodeC(branchAddress, PC - branchAddress);
}
```

replace parameter c (pc-relative address) of branch instruction @ branchAddress with PC-branchAddress

(meaning: jump to pc (current instruction)

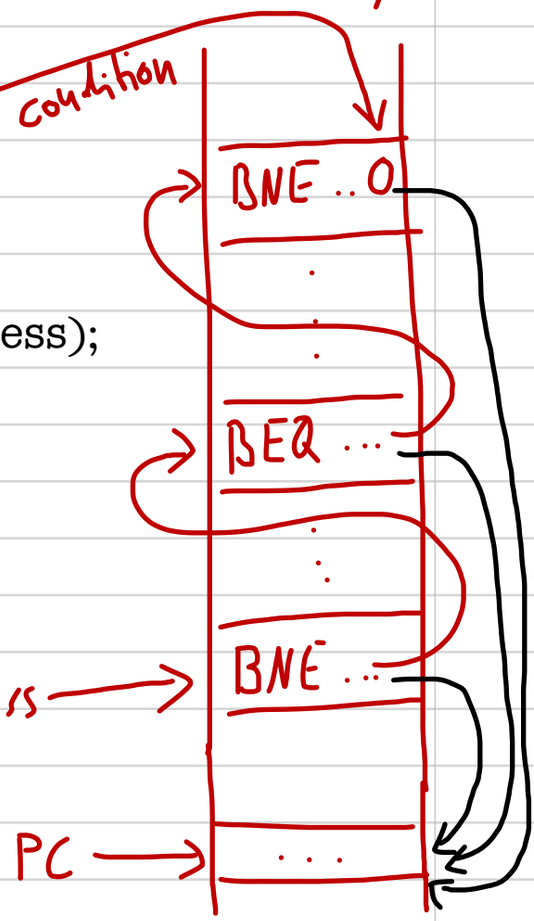
```
fixLink(int branchAddress) {
 int nextBranchAddress;

 while (branchAddress != 0) {
   nextBranchAddress = decodeC(branchAddress);
   fixUp(branchAddress);
   branchAddress = nextBranchAddress;
 }
}
```

termination condition

get parameter c

(list head) branchAddress



while Statement

```
while ( x == y )        CMP leftItem->reg, leftItem->reg, rightItem->reg
 {                      BNE leftItem->reg, 0, leftItem->fls
  ...                   ...
 }                      BR 0, 0, bJumpAddress - PC
                        ...
```

```
whileStatement() {
 struct item_t* item;
 int bJumpAddress;

 if (symbol == WHILE) getSymbol(); else error("while expected");
 if (symbol == LPAREN) getSymbol(); else error("missing '('");

 bJumpAddress = PC;

 item = malloc(sizeof(struct item_t));
 expression(item);

 if (item->type == BOOL_TYPE) {
  loadBool(item);
  cJump(item);
  fixLink(item->tru);
 } else error("boolean expression expected");

 if (symbol == RPAREN) getSymbol(); else error("missing ')'");

 if (symbol == BEGIN) {
  getSymbol();
  statementSequence();
  if (symbol == END) getSymbol(); else error("missing '}'");
 } else statement();

 bJump(bJumpAddress);
 fixLink(item->fls);
}

bJump(int backAddress) {
 put(BR, 0, 0, backAddress - PC);
}
```

later for Boolean expressions

jump here if comparison evaluates to false!