Translation flow controls with the optimization

WANG Hanfei

November 27, 2017

Contents

2015 级弘毅班编译原理课程设计第七次编程练习

translation flow control construction with elimination redundant goto.

1 semantic rules

`B.true` : the inherited attribute, the label where the true exit gone.
`B.false`: the inherited attribute, the label where the false exit gone. (see the implementation source file `control.zip` included)
If one of the `true` or `false` exit label is just immediately stmt after constrol intruction like:

```
    if (a > b) then goto L1
    goto L2
L1: stmt
L2:
```

or

```
    if (a > b) then goto L1
    stmt
L1: goto L2
L2:
```

the redundant goto stmt can be eliminated by transforming the stmts as follow:

```
    ifNOT (a > b) then goto L2
    stmt
L2:
```

or

```
    if (a > b) then goto L2
    stmt
L2:
```

(**see dragon book pp 405**)
Example:

```
if (a > b and c > d) then s1 else s2
```

if we translate boolean expression "`(a > b) and (c > d)`" with the elimination of the redundant goto. we must know the struct of the stmt followed by "`(a > b) and (c > d)`". Suppose that is:

```
    s1.code
    goto L
L1: s2.code
L:
```

so the true exit of `(a > b and c > d)` is immediately followed. this info tell `(c > d)`, it should translate in the inverse for the seamless connection:

```
    ifNOT (c > d) goto L1
    s1.code
    goto L
L1: s2.code
L:
```

and we get again the code chain with true exit immediately followed. so it tell for our last expression the true exit is immediate so `(a > b)` will be

```
    ifNOT (a > b) goto L1
    ifNOT (c > d) goto L1
    s1.code
    goto L
L1: s2.code
L:
```

if we change "`and`" to "`or`" in the above stmt. in the last step of translation, the immediate stmt of `(a > b)` will be the `false` exit. so: `if (a > b or c > d) then s1 else s2` will be translate to:

```
    if (a > b) goto L2
    ifNOT (c > d) goto L1
L2: s1.code
    goto L
L1: s2.code
L:
```

This suggests us to introduce a new inherited attribute value "`fall`" for `B.true` and `B.false`, which means the `true` or `false` exit is immediate stmt after current control stmt.

So "`B.true = fall`" means the `true` exit is immediate stmt, don't need the `true` exit label if constrol translate in the inverse form "`ifNOT`". "`B.false = fall`" means the false exit is immediate stmt, don't need the `false` exit label if control translate directely and `false` goto can be eliminated.

the new semantic rules are as follow:

```
production              |  semantic rules
------------------------+------------------------------------------
B -> id1 relop id2      |  if ((B.true != fall) && (B.false != fall)) then
                        |    B.code = gen("if ", id1.name, relop.name,
                        |                 id2.name, "goto", B.true) ||
                        |             gen("goto ", B.false)
                        |  else if (B.false != fall) then
                        |    B.code = gen("ifnot", id1.name, relop.name,
                        |                 id2.name, "goto", B.false)
                        |    else if (B.true != fall) then
                        |      gen("if ", id1.name, relop.name, id2.name,
                        |          "goto", B.true)
                        |     else gen ("")
------------------------+----------------------------------------------
B -> B1 or B2           |  if (B.true == fall) then B1.true = newlabel()
                        |  else B1.true = B.true
                        |  B1.false = fall
                        |  B2.true = B.true
                        |  B2.false = B.false
                        |  B.code = if (B.true !=fall) then
                        |    B1.code || B2.code
                        |  else B1.code || B2.code || B1.true:
------------------------+----------------------------------------------
B -> B1 and B2          |  B1.true = fall
                        |  if (B..false == fall) then B1.false = newlabel()
                        |  else B.false
                        |  B2.true = B.true
                        |  B2.false = B.false
                        |  B.code = if (B.false != fall) then
                        |    B1.code || B2.code
                        |  else B1.code || B2.code || B1.false
------------------------+----------------------------------------------
B -> not B1             |  B1.true = B.false
                        |  B1.false = B.true
                        |  B.code = B1.code
------------------------+----------------------------------------------
S -> if (B) then S1     |  B.true = fall
                        |  B.false = S.next
                        |  S1.next = S.next
                        |  S.code = B.code || S1.code
------------------------+----------------------------------------------
S -> if (B) then S1     |  B.true = fall
     else S2            |  B.false = newlabel()
                        |  S1.next = S.next
                        |  S2.next = S.next
                        |  S.code = B.code || S1.code || gen ("goto" S.next)
                        |           || B2.false || S2.code
------------------------+----------------------------------------------
S -> while (B) S1       |  B.true = fall
                        |  B.false = S.next
                        |  S1.next = newlabel()
                        |  S.code = S1.next || B.code || S1.code ||
                        |           gen ("goto" S1.next)
------------------------+----------------------------------------------
S -> repeat S1 until B  |  B.true = S.next
                        |  B.false = newlabel
                        |  S1.next = newlabel
                        |  S.code = B.false || S1.code || S1.next || B.code
------------------------+----------------------------------------------
S -> S1 S2              |  S1.next = newlabel()
                        |  S2.next = S.next
                        |  S.code = S1.code || S1.next || S2.code
-----------------------------------------------------------------------
```

2 transform inherited att to synthesize att: case analysis

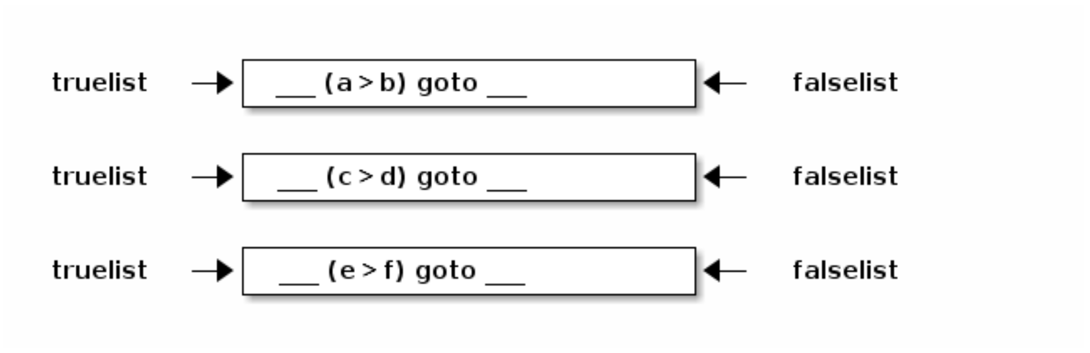
To implement the SDD above, we use the backpatching by change inherited attribute `true`, `false` and `next` to `truelist`, `falselist` and `nextlist`. and backpatching not only the goto label, but also the first blank of relation stmt with "`if`" or "`ifnot`".

To do so, we first translate the relation expression `(x RELOP x)` to "`_1_ (x RELOP y) goto _2_`" with 2 blanks, and fill the first blank with "`if`" or "`ifnot`" and the second with a label by the followed contexts.

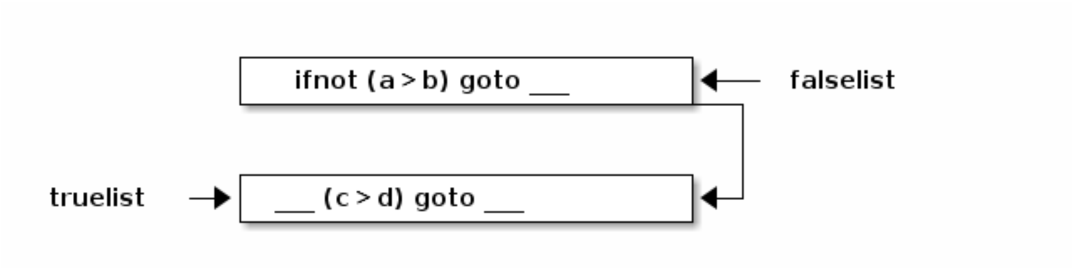
2.1 Ex1:

```
if  a > b and c > d or e > f then x := 1
```

- every relop will translate initialy with 2 blanks stmt. and the truelist and falselist points to it.

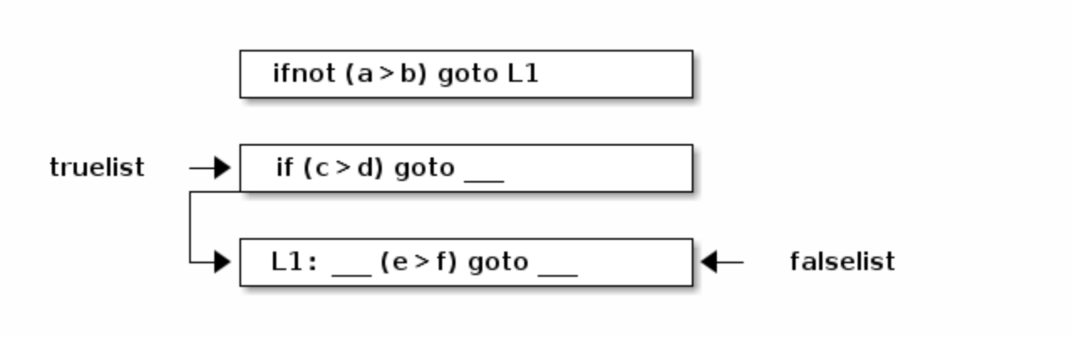


- compose "`a > b and c > d`"



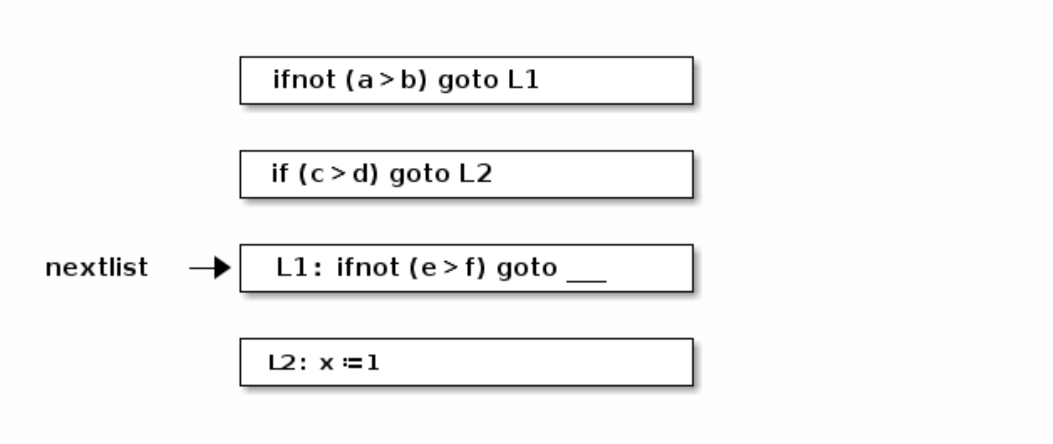
```
/* each the composition of relop, we can always backpatching
   the first blank of the first operand, if the first is just
   single relop stmt, and keep always truelist and falselist
   with the same tail. and that is always only stmt which
   has the first blank required to fill if or ifnot! */
```

- compose "`a > b and c > d or e > f`"



```
/* if the first operand is the composition of relop, we can
   always backpaching the first blank of the tail of
   the truelist or false_list of the first operand, and return
   the code chain satisfies the properties descripted in 2/ */
```

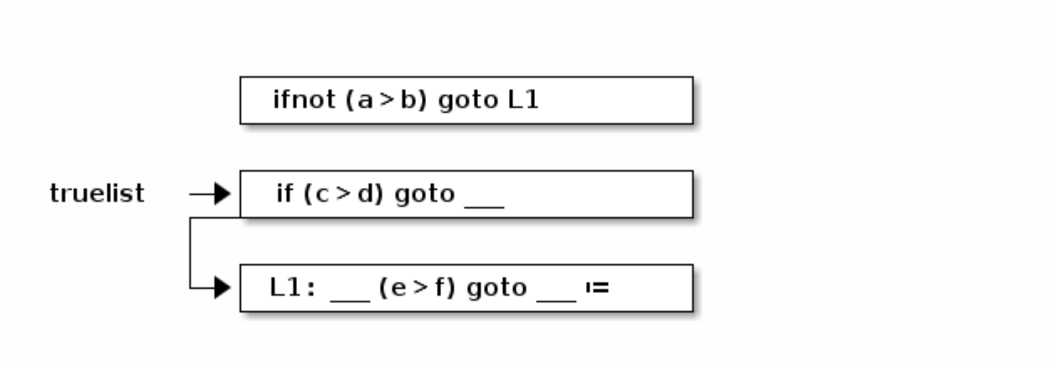
- compose "`if a > b and c > d or e > f then x := 1`"



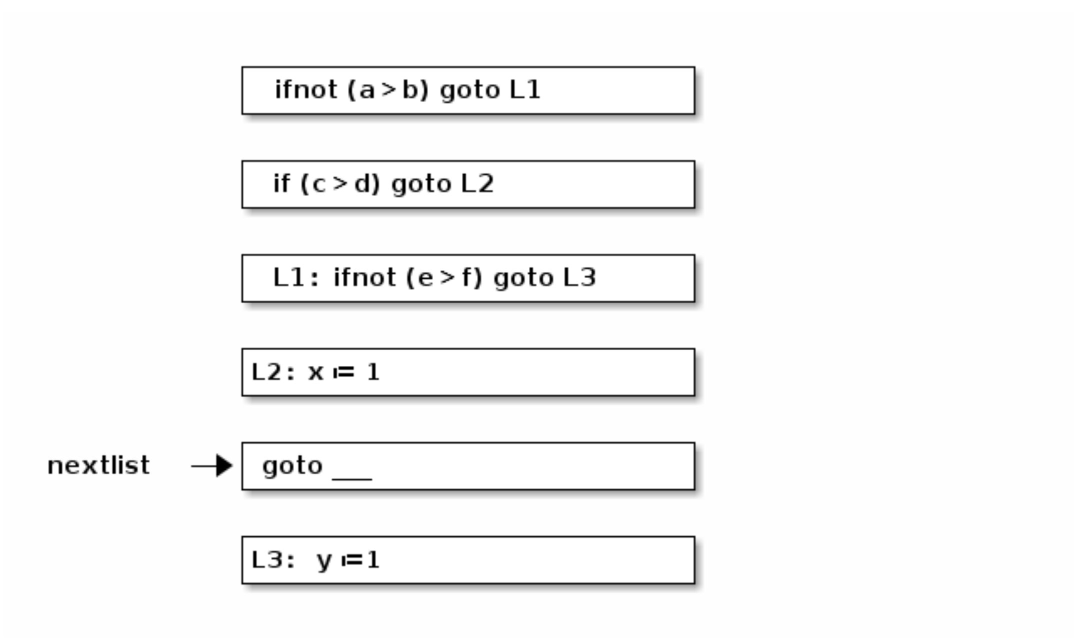
2.2 Ex2:

`if a > b and c > d or e > f then x := 1 else y := 1`

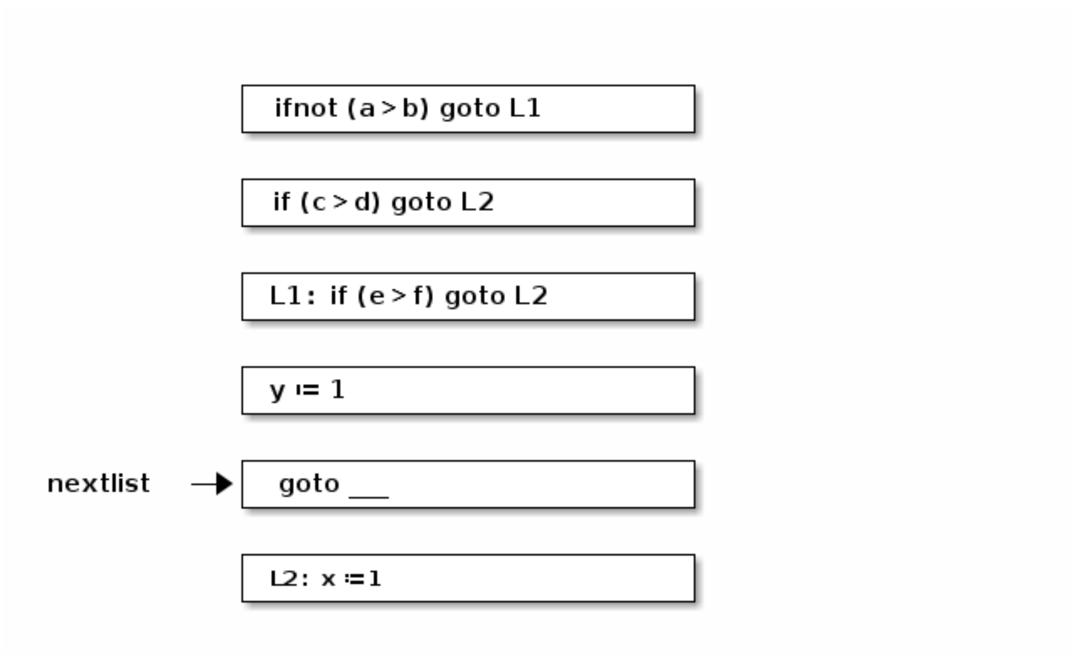
- "`a > b and c > d or e > f`" will return:



- we can directly translate as:



```
/* if we change the true branch and the false branch, we only
   need 2 label. like: */
```



- We will observe how do the inversion of true and false branch.
  if the condition expression is just a single operand like:

```
if a > b then x := 1 else y := 1.

   ##code _listing:
           ifnot (a > b) goto  l0
           x := 1
           goto l1
   l0:     y := 1
   l1:
   ##end_listing
```

or

```
if a > b and c > d then x := 1 else y := 1.

   ##code _listing:
           ifnot (a > b) goto  l0
           ifnot (c > d) goto  l0
           x := 1
           goto l1
   l0:  y := 1
   l1:
   ##end_listing
```

so the `truelist` is immediate to `x := 1` (or `B.true = fall`), we does not need generate a newlabel for `x := 1`.

such case occur only when `truelist` head and truelist tail are the same code (only one element in the `truelist`).

respectly, there are also the case where `falselist` head and tail are the same element (or `B.false = fall`), such as:

```
if a > b or c > d  then x := 1 else y := 1.
```

which can be translated as:

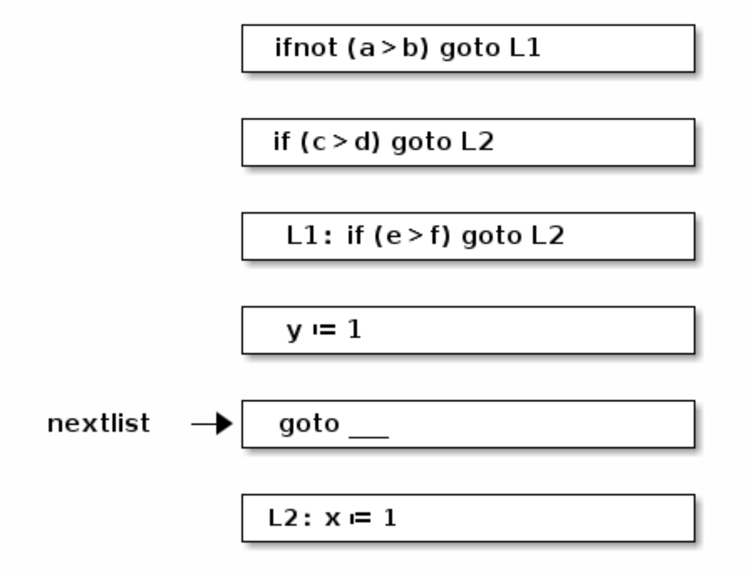
```
##code _listing:
        if (a > b) goto  l0
        if (c > d) goto  l0
        y := 1
        goto l1
l0:     x := 1
l1:
##end_listing
```

this improvement will earn a label, in the tradeoff the execution orders.

so `is_same_head_tail(list)` is introduced to do the above test.

for our example, `is_same_head_tail(falselist) = true`.

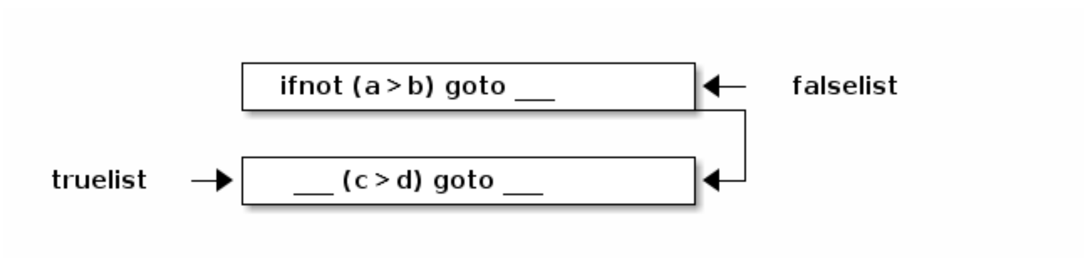
- so



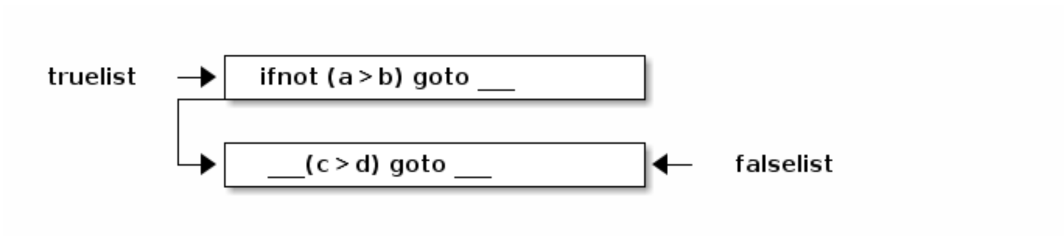
2.3 Ex3:

`if  not(a > b and c > d) or (e > f) then x := 1 else y:=1.`

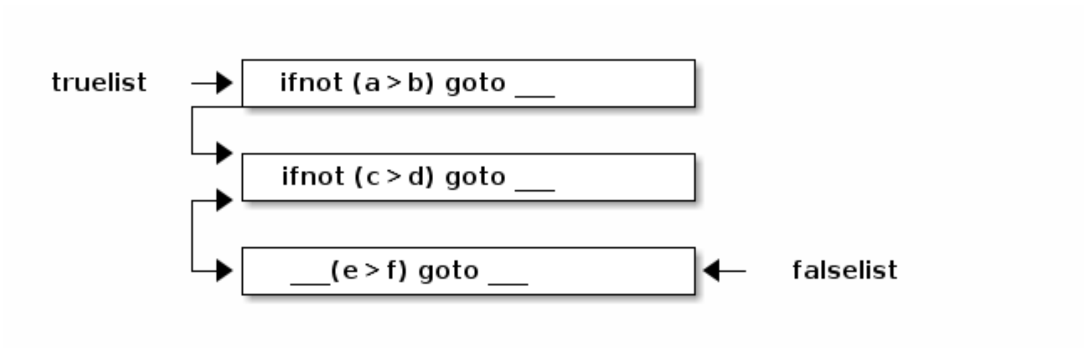
- compose "`a > b and c > d`"



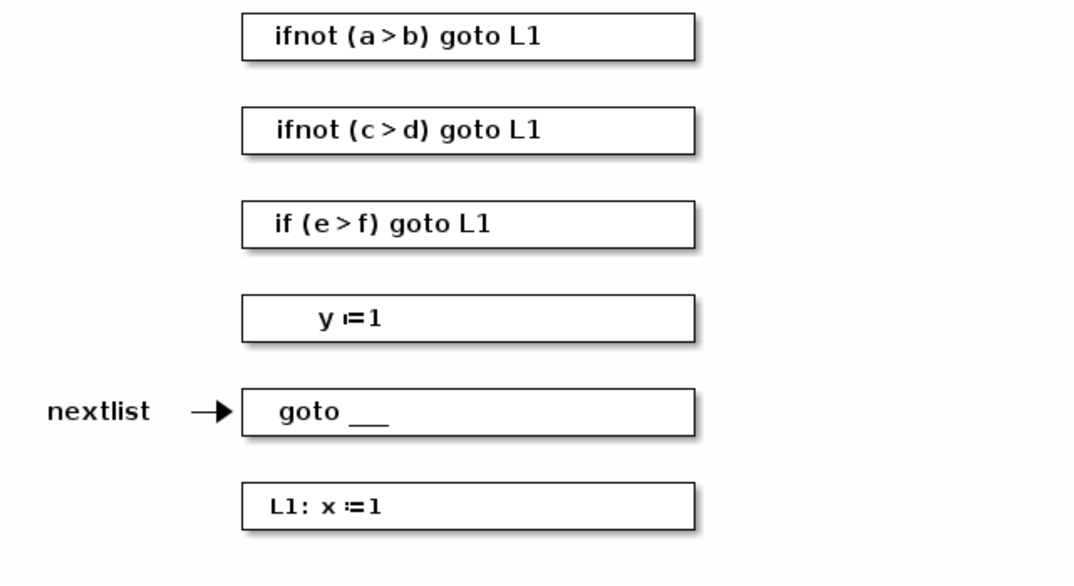
- transform "`not (a > b and c > d)`": swap truelist and falselist.



- compose "`not(a > b and c > d) or (e > f)`"



- `is_same_head_tail(falselist) = true`.
- chaining true exit and false exit in reverse order.

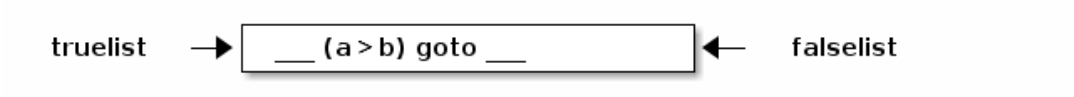


2.4 Ex4.:

`if not a > b then x := 1.`

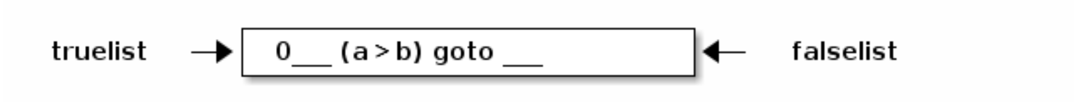
- `not a > b`.

  because the operand will generate the truelist and falselist which point to same stmt for the negation of a single relation operation.

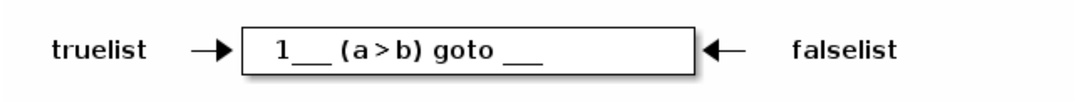


So the changing truelist to falselist which does not help us to express the negation.

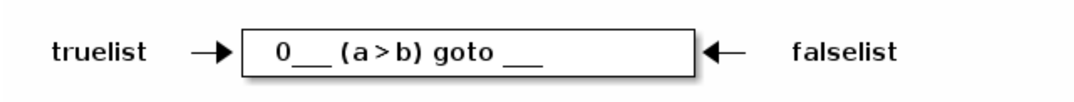
so we must have an extra attribute to expression if the operation is positive or negative form. To do so, we just "1" or "0" to the generated code as the prefix to express it, like:



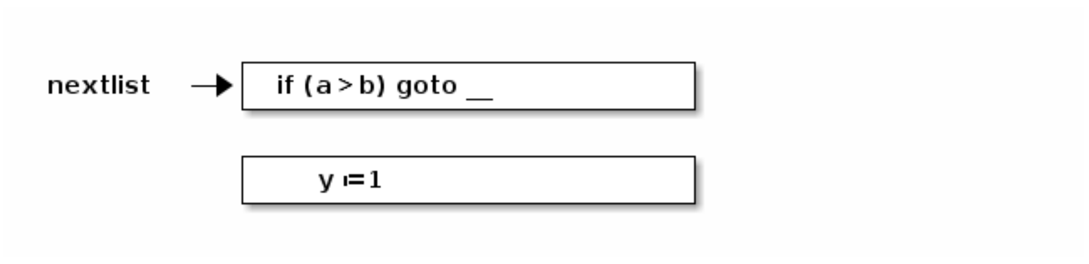
so the operation will take the negative form, and for the positive form, it will be:



- so `is_negative(stmt)` is introduced. and for `not a > b`:



we have `is_same_head_tail(truelist) == TRUE` and `is_negative(truelist) == TRUE`. and "`if B then x:=1.` " tell us the `truelist` exit is immediately followed by "`x := 1`". so



3 summaries of the case analyisis

- for a boolean expression, we always keep the `truelist` and `falselist` with the same tail. and we have always either

`is_same_head_tail(truelist) == FALSE`

or

`is_same_head_tail(flaselist) == TRUE`

the tail of the lists is the only one stmt that require fill the head with `if` or `ifnot`.

- `is_negative(stmt)` tell us the operand is positive or negative form.
- if `B1` is first boolean expression, and `B2` is a boolean expression that will joint `B1` with a relation `relop` (`B1 relop B2`). then we have 2 possibilities of chaining `B1.code` and `B2.code`

– fill all `B1.truelist` except the tail with the begin label of `B2.code`. and `merge` two `falselist`. (for `relop AND`) and we denote this mode by `FILL_TRUE`
– fill all `B1.falselist` except the tail with the begin label of `B2.code`. and `merge` two `truelist`. (for `relop OR`) and we denote this mode by `FILL_FALSE`
the tail of B1.truelist will fill the head with

relop	fill mode	`is_negative`	the head to fill
and	`FILL_TRUE`	`FALSE`	`ifnot`
and	`FILL_TRUE`	`TRUE`	`if`
or	`FILL_FALSE`	`FALSE`	`if`
or	`FILL_FALSE`	`TRUE`	`inot`

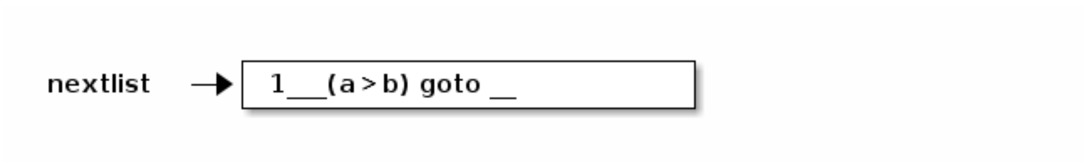
we implement this combination by a function:

`~ATT combine(ATT att1, ATT att2, int mode)~`

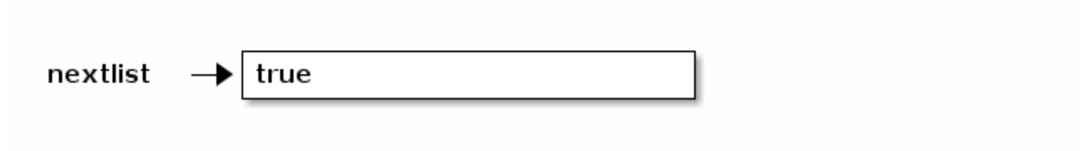
(see `control.y` for detail)

- boolean constant `true` or `false`

for boolean constant "`true`" and "`false`", generate "`true`" or "`false`" as the stmt. and each time of synthesis of relop, we will test if an operand is boolean constant by `is_boolean_true ()` or `is_boolean_false()`. if the case, we just return the simplified boolean expression as tranlation result.
– Ex: `not (a > b) and true`



– `(a > b) or true`



`truelist` and `falselist` will be empty

4 new semantic rules

```
production               |  semantic rules
-------------------------+---------------------------------------------
B -> id1 relop id2       |  B.code = gen ("1(", id1.name, relop.name,
                         |                id2.name, ")")
                         |  /* 1 will indicate that is positive form */
                         |  B.truelist = & B.code
                         |  B.falselist = & B.code
-------------------------+---------------------------------------------
B -> true                |  B.code = "true"
                         |  B.truelist = NULL;
                         |  B.falselist = NULL;
-------------------------+---------------------------------------------
B -> false               |  B.code = "false"
                         |  B.truelist = NULL;
                         |  B.falselist = NULL;
-------------------------+---------------------------------------------
B -> B1 or B2            |  if (B1.code == "true" || B2.code == "true")
                         |    B.code = "true";
                         |    B.truelist = B.falselist = NULL;
                         |  if (B1.code == "false")
                         |    B.att = B2.att /* every attribute of B */
                         |  if (B2.code == "false")
                         |    B.att = B1.att;
                         |
                         |  B.att = combine (B1.att, B2.att, FILL_FLASE)
-------------------------+----------------------------------------------
B -> B1 and B2           |  if (B1.code == "false" || B2.code == "false")
                         |    B.code = "false";
                         |    B.truelist = B.falselist = NULL;
                         |  if (B1.code == "true")
                         |    B.att = B2.att /* every attribute of B */
                         |  if (B2.code == "true")
                         |    B.att = B1.att;
                         |
                         |  B.att = combine (B1.att, B2.att, FILL_TRUE)
-------------------------+---------------------------------------------
B -> not B1              |  if (B1.code == true) B.code = false
                         |  if (B1.code == false) B.code = true
                         |  B.truelist = B1.falselist
                         |  B.falselist = B1.truelist
                         |  if (B1 is single operand)
                         |    B1.code[0] = B1.code[0]== 1? 0 : 1
                         |    /* change positive form to negative,
                         |       or inverse */
                         |  B.code = B1.code
-------------------------+---------------------------------------------
S -> if (B) then S1      |  S.att = translate_if_then(B.att, S1.att)
                         |  /* to do */
-------------------------+---------------------------------------------
S -> if (B) then S1      |  S.att = translate_if_then_else (B.att, S1.att
     else S2             |                                   S2.att)
                         |  /* to do */
-------------------------+---------------------------------------------
S -> while (B) S1        |  S.att = translate_while(B.att, S1.att)
                         |  /* to do */
-------------------------+---------------------------------------------
```

```
S -> repeat S1 until B  |  S.att = translate_repeat(S1.att, B.att)
                        |  /* to do */
-----------------------------------------------------------------------
```

5 TODO

5.1 implement the following function

(in `control.y`)

```
ATT translate_if_then (ATT cond, ATT s);
ATT translate_if_then_else (ATT cond, ATT s1, ATT s2);
ATT translate_repeat(ATT body, ATT cond);
ATT translate_while (ATT cond, ATT body);
```

where `ATT` is defined as:

```
typedef struct att {
  CODE * true_list;      /* For E.truelist and S.nextlist */
  CODE * false_list;
  CODE * code;
  CODE * break_list;     /* add for the break statement */
  CODE * continue_list;  /* add for continue statement */
} ATT;
```

and `CODE` is defined as

```
typedef struct code {
  char * code;
  char * label; /* for goto label, used also
                   for backpatching list pointer */
  struct code * next;
} CODE;
```

5.2 More optimization:

if there is a `break` in `if_then` or `if_then_else like`:

```
while a > b do begin if c > d then break; x:=1 end.
```

we should translate as:

```
##code _listing:
l1:     ifnot (a > b) goto  l2
        ifnot (c > d) goto  l0
        goto l2
l0:     x := 1
        goto l1
l2:
##end_listing
```

but we can just `merge` the "`c > d`" `truelist` to the `while nextlist`, so we can eliminate a redundant `goto` like:

```
l0:     ifnot (a > b) goto  l1
        if (c > d) goto  l1
        x := 1
        goto l0
l1:
```

(see `control1.exe` (DOS) or `control1.bin` (Linux) for the output)

Can you implement `translate_if_then()` and `translate_if_then_else()` with this consideration?

5.3 提交方式

please send your `control.y` as attached file to mailto:hfwang@whu.edu.cn?subject=ID(07) where the `ID` is your student id number.

–hfwang November 27, 2017