CSc 453

Compilers and Systems Software

16 : Intermediate Code IV

Department of Computer Science
University of Arizona

collberg@gmail.com

Control Structures

Control Structures

```
IF E THEN        WHILE E DO              REPEAT     LOOP
  S                S                       S          S
ENDIF            ENDDO;                  UNTIL E;   ENDLOOP;
IF E THEN S      IF E1 THEN S1           <Name>:  LOOP S1;
ELSE S           ELSIF E2 THEN S2          EXIT <Name> WHEN E;
ENDIF;           ELSE S3                   S2;
                 ENDIF                   ENDLOOP;
FOR i:INT := E1 TO E2 BY E3 DO           FOR i IN [E1, E2, ...] DO
  S                                        S
ENDFOR;                                  ENDFOR;
```

Boolean Expressions

Short Circuit Evaluation

- With short circuit evaluation of boolean expressions we only evaulate as much of the expression as is necessary to determine if the expression evaulates to true or false.
- Pascal does not have short-circuit evaluation. Many Pascal programmers have been burnt by this type of code:

  ```
  if p <> nil and p^.data = 32 then ...
  ```
- On the other hand, Modula-2 (which only supports short-circuit evaluation) sometimes get burnt when a function with side-effects doesn't get executed:

  ```
  if a < t and (f(45) < 10) then ...
  ```
- Some languages (Ada, Algol) let the programmer decide when to use short-circuit evaluation.

Boolean Expressions

E ::= E 'OR' E | E 'AND' E | 'NOT' E |
'(' E ')' |
E relop E |
'true' | 'false'

relop ::= '<' | '<=' | '=' | '<>' | '>=' | '>'

Language Design Space:

- Short-circuit evaluation of AND & OR?

```
if p <> nil and p^.data = 32 then ...
if a < t or (f(45) < 23) then ...
```

Compiler Design Space:

- Numerical or flow-of-control representation?

Numerical Representation

Numerical Representation

- The main advantage of implementing boolean expressions using a numerical representation is that it is very easy to implement.
- We simply extend our arithmetic expressions with new operators for AND, OR, NOT, and the relational operators.

Numerical Representation

- Boolean expressions are evaluated similarly to arithmetic expressions.

Operators:

FALSE	$\equiv$	0
TRUE	$\equiv$	*any value* > 0
x **AND** y	$\equiv$	$x * y$
x **OR** y	$\equiv$	$x + y$
NOT x	$\equiv$	**IF** $x = 0$ **THEN** t := 1 **ELSE** t := 0;
$x < y$	$\equiv$	**IF** $x < y$ **THEN** t := 1 **ELSE** t := 0;

Numerical Representation. . .

```
B := X < 10;
WHILE X > 5 DO
  DEC (X);
ENDDO
```

```
100:IF X < 10 GOTO 103
101:B := 0
102:GOTO 104
103:B := 1
104:IF X > 5 GOTO 107
105:t1 := 0
106:GOTO 108
107:t1 := 1
108:IF t1 = 0 GOTO 111
109:X := X - 1;
110:GOTO 104
111:
```

Flow-of-Control Representation

Flow-of-Control

- The value of a boolean expression is given by our position in the program.
- The value of `X<10` is given by position 103 (if `X<10=TRUE`) or 101 (if `X<10=TRUE`).

```
B := X < 10;
WHILE X > 5 DO
  DEC (X);
ENDDO
```

```
100:IF X < 10 GOTO 103
101:B := 0
102:GOTO 104
103:B := 1
104:IF X <= 5 GOTO 107
105:X := X - 1;
106:GOTO 104
107:
```

Short-Circuit Evaluation

Short-Circuit Code

- What happens if the function `f` has side-effects (e.g. if it changes the value of a global variable)? Well, in such cases short-circuit code will have different semantics from the non-short-circuit code.
- In this example we use flow-of-control for the short-circuit evaluation, and numerical representation for the full evaluation. We could have given both examples using flow-of-control.

Short-Circuit Code – **AND**

```
IF (X > 5) AND f(34) THEN
    DEC (X);
ENDIF
```

Short Circuit	Full Evaluation
100: **IF** X <= 5 **GOTO** 103	100: t_1 := X > 5;
101: **IF NOT** f(34) **GOTO** 103	101: t_2 := f(34);
102: X := X - 1;	102: t_3 := t_1 **AND** t_2;
103:	103: **IF NOT** t_3 **GOTO** 105
	104: X := X - 1;
	105:

Short-Circuit Code – OR

```
IF (X > 5) OR f(34) THEN
   DEC (X);
ENDIF
```

Short Circuit	Full Evaluation
100: **IF** X > 5 **GOTO** 103	100: t_1 := X > 5;
101: **IF** f(34) **GOTO** 103	101: t_2 := f(34);
102: **GOTO** 104	102: t_3 := t_1 **OR** t_2;
103: X := X - 1;	103: **IF NOT** t_3 **GOTO** 105
104:	104: X := X - 1;
	105:

Implementing Control Structures

Control Structures

- One problem we're faced with when generating code for control structures, is that we generate code for the boolean expression before we generate code for the statements. Hence, when we generate jumps from out of (a complex) boolean expression we don't know where to jump to.
- Each expression is given two slots (`true & false`) which are filled in with the location to which we will jump if the expression is true or false respectively.

Control Structures. . .

- Each statement also has a slot `next` which is the location of the instruction following the statement. This is used when we want to jump out of the body of a control statement.
- Using the **next** label avoids some jumps-to-jumps. Consider the example in the slide. When we jump out of the inner IF-statement (when the expression evaluates to **false**) we jump to the instruction following the IF. That happens to be a jump back to the top of the WHILE-statement. Using the **next**-slot fixes this.

Implementing Control Structures

- We generate code for the expression before the body of the control structure. How can we know where to jump?
- We give each expression two slots which get filled in when the appropriate label is known:

  true (false) Where to jump to when the expression is true (false).
- We give each statement one slot **next** which gets filled with the label of the next statement when it becomes known.

```
WHILE x<b DO          ⇒   L1: if x >= b goto L2
  IF a = 1 THEN                if a <> 1 goto L4
    X := 1;                    X := 1
  ENDIF                   L4: goto L1
ENDDO                     L2:
```

Semantic Attributes

true
: An **inherited** attribute passed into boolean expressions. Holds the label to which the expression should jump if it evaluates to **true**.

false
: Where to jump to if the expression evaluates to **false**.

next
: An **inherited** attribute passed into statements. Holds the label of the statement following the current one.

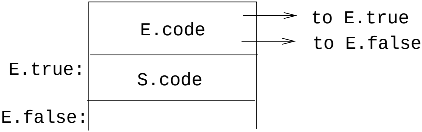
begin
: A **synthesized** attribute that holds the label of the beginning of a WHILE-statement.

IF Statements

- We first generate code for the expression, then the body of the loop.
- We put a label (E.true) at the beginning of the loop body. We jump to this label from every place within the expression where we can determine that it is evaluated to true (there may be several such places).
- Similarly, we add a label after the end of the statement (E.false) to which we jump when the statement evaluates to false.

Control Structures –IF

S ::= 'IF' E 'THEN' SS_1 'ENDIF'



Example:

```
IF (X > 5) THEN
   DEC (X);
ENDIF
```

$\Downarrow$

```
100:  IF X > 5 GOTO 102 (=E.true)
101:  GOTO 103 (=E.false)
102:  X := X - 1;
103:
```

Control Structures –**IF**

- The implementation is encoded into these attribute evaluation rules.
- We start by creating the new label E.`true`.
- Then we set E.`false` to be the same label as S.`next`. This means that if the expression evaluates to false, then we will jump to the statement immediately following the IF-statement, which is exactly what we want to do.

Control Structures –**IF**. . .

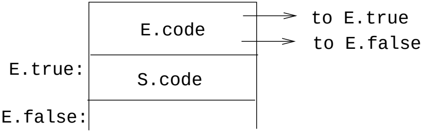
- The next rule (S_1`.next := `S`.next;`) says that: “if we should need to jump out of the body of the IF-statement, then we should jump to the statement immediately following the current statment”.
- Finally, we have the code generation rule which says that the code generated from an IF-statement consists of the code for the expression, the label E`.true`, and the code for the statement body.
- Normally we don’t generate code as part of the attribute grammar, but it is certainly possible to do so.

Control Structures –**IF**...

S ::= 'IF' E 'THEN' SS$_1$ 'ENDIF'

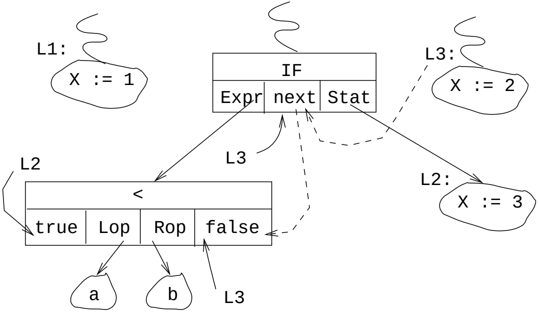
S ::= E S_1

{ E.true := newlabel();
  E.false := S.next;
  S_1.next := S.next;
  S.code := E.code ‖
    'E.true ":"' ‖
    S_1.code;
}



Control Structures –**IF**. . .

```
X:=1;
IF a<b THEN X:=3 ENDIF;
X:=2;
```




```
IF a < b GOTO L2
GOTO L3
```

Control Structures –**IF**...

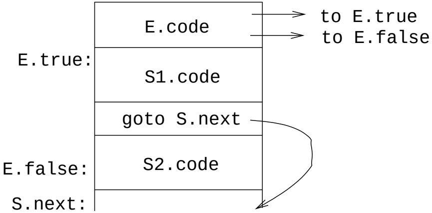
```
 L1:      X := 1
          if a < b then goto L2
          goto L3
L2:       X := 3;
L3:       X := 2;
```

$S ::= E\ S_1$

$\{\quad E$.true := newlabel();
E.false := S_1.next := S.next;
S.code := E.code $\|$
'E.true ":"' $\|$ S_1.code; $\}$

Control Structures –**IF-ELSE**

S ::= 'IF' E 'THEN' SS_1 'ELSE' SS_2 'ENDIF'



Control Structures –**IF-ELSE**. . .

Example:

```
IF (X > 5) THEN DEC (X);
ELSE INC (X); ENDIF
              ⇓
100:  IF X > 5 GOTO 102 (=E.true)
101:  GOTO 104 (=E.false)
102:  X := X - 1;
103:  GOTO 105 (=S.next)
104:  X := X - 1;
```

Control Structures –**IF-ELSE**

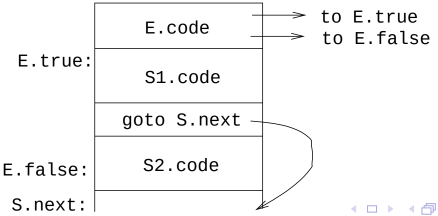
- We have to generate two new labels, one attatched to the beginning of the THEN-part the other to the beginning of the ELSE-part. These are then passed into the expression to make sure we jump to the right places.
- If we need to jump out of the THEN-part or the ELSE-part we should land at the statement immediately following the current statment, hence we set S_1`.next` and S_2`.next` to S`.next`

Control Structures –IF-ELSE...

S ::= 'IF' E 'THEN' SS_1 'ELSE' SS_2 'ENDIF'

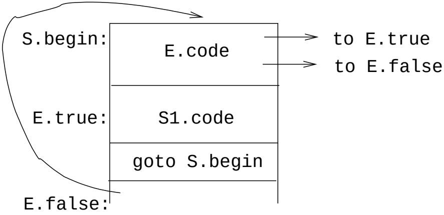
S ::= E S_1 S_2

{ E.true := newlabel();
E.false := newlabel();
S_1.next := S.next;
S_2.next := S.next;
S.code := E.code ‖ 'E.true ":"' ‖ S_1.code ‖
'goto S.next' ‖ 'E.false ":"' ‖ S_2.code; }



Control Structures –WHILE

S ::= 'WHILE' E 'DO' SS_1 'ENDDO'



Example:

WHILE (X>5) **DO** DEC(X); **ENDDO**

$\Downarrow$

100: **IF** X > 5 **GOTO** 102 (=E.true)
101: **GOTO** 104 (=E.false)
102: X := X - 1;
103: **GOTO** 100 (=S.begin)
104:

Control Structures –**WHILE**

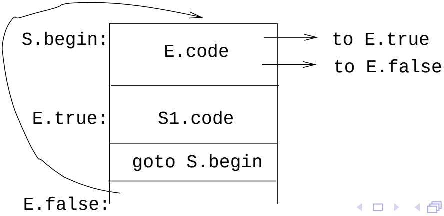
- S.`begin` is a label we create and attatch to the expression itself. Later we will complete the loop by generating a jump back to this label.
- E.`true` is a label we attatch to the loop body.
- E.`false` is set to the instruction following the loop; this is where we jump if the loop condition evaluates to false.

Control Structures –**WHILE**. . .

S ::= 'WHILE' E 'DO' SS_1 'ENDDO'

$S ::= E\ S_1$

$$
\{\quad
\begin{array}{lll}
S\texttt{.begin} & := & \texttt{newlabel();} \\
E\texttt{.true} & := & \texttt{newlabel();} \\
E\texttt{.false} & := & S\texttt{.next;} \\
S_1\texttt{.next} & := & S\texttt{.begin;} \\
S\texttt{.code} & := & \texttt{'}S\texttt{.begin ":"'} \parallel E\texttt{.code} \parallel \texttt{'}E\texttt{.true ":"'} \parallel \\
 & & \quad S_1\texttt{.code} \parallel \texttt{'goto } S\texttt{.begin'}
\end{array}
\}
$$



Control Structures – RelOp/**NOT**/**AND**/**NOT**

- The boolean connectives AND, OR, and NOT, are interesting since they don't require us to generate any new code; all we have to do is to assign the correct labels to the `true` and `false` attributes.
- All we have to do for the NOT operator is to switch the true and false attributes! In other words, NOT means that we jump to the true label when the expression evaluates to false and to the false label when the expression evaluates to true.

Control Structures – RelOp/**NOT**

Relational Operators:

E ::= E_1 < E_2

E_0 ::= $E_1 < E_2$

```
{E.code :=
  't1 := E1.code' ||
  't2 := E2.code' ||
  'IF t1 < t2 GOTO E0.true' ||
  'GOTO E0.false'; }
```

NOT:

E ::= 'NOT' E_1

E_0 ::= **NOT** E_1

```
{  E1.true  := E0.false;
   E1.false := E0.true; }
```

Control Structures –**OR**

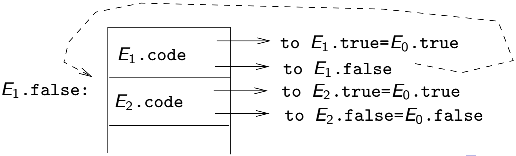
- Since we're assuming short-circuit evaluation when the left-hand-side expression evaluates to false, we have to jump to the right-hand-side expression and evaluate that one too.
- If either one of the expressions evaluates to true then the complete expression is true also. Therefore we set E_1`.true` and E_2`.true` to the same label as E_0`.true`.

Control Structures –OR

E ::= E_1 'OR' E_2

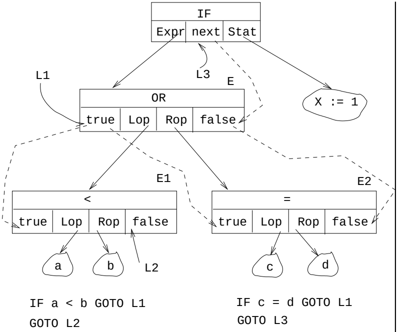
E_0 ::= E_1 **OR** E_2

```
{   E1.true  := E0.true;
    E1.false := newlabel();
    E2.true  := E0.true;
    E2.false := E0.false;
    E.code   := E1.code ||
                'E1.false ":"' || E2.code
}
```



Example – OR

IF (a < b) **OR** c = d **THEN**
    X := 1
ENDIF



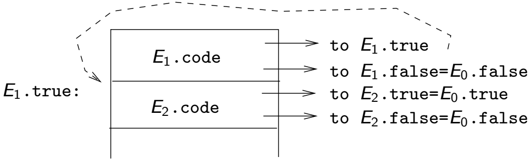

```
    IF a < b GOTO L1
    GOTO L2
L2: IF c = d GOTO L1
    GOTO L3
L1: X := 1;
L3:
```

Control Structures –AND

E ::= E_1 'AND' E_2

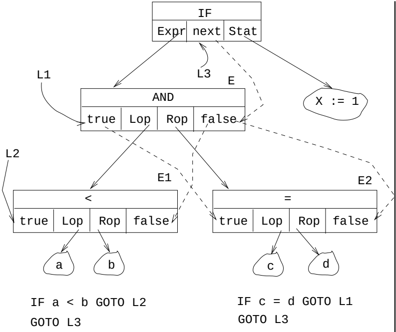
E_0 ::= E_1 **AND** E_2

$\{$ E_1.true := newlabel();
E_1.false := E_0.false;
E_2.true := E_0.true;
E_2.false := E_0.false;
E.code := E_1.code $\|$ 'E_1.true ":"' $\|$ E_2.code
$\}$



Example – AND

IF (a < b) **AND** c = d **THEN**
    X := 1
ENDIF

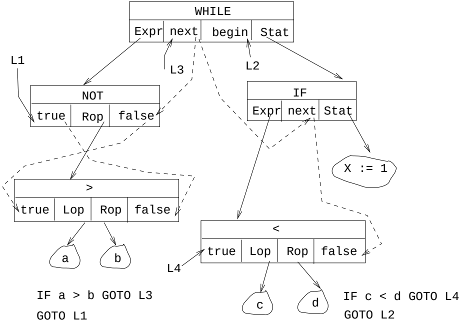



```
    IF a < b GOTO L2
    GOTO L3
L2: IF c = d GOTO L1
    GOTO L3
L1: X := 1;
L3:
```

Examples

Example – WHILE

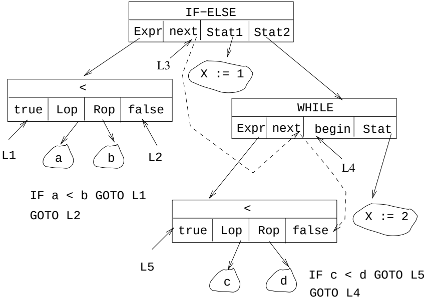
```
WHILE NOT (a > b) DO
   IF c<d THEN X:=1 ENDIF
ENDDO
```




```
    IF a > b GOTO L3
    GOTO L1
L1: IF c = d GOTO L4
    GOTO L2
L4: X := 1;
    GOTO L2
L3:
```

Example – IF-ELSE

IF a < b **THEN** X := 1
ELSE **WHILE** c<d **DO** X:=2; **ENDDO**
ENDIF




```
      IF a < b GOTO L1
      GOTO L2
L1:   X := 1
      GOTO L3
L2:
L4:   IF c < d GOTO L5
      GOTO L3
L5:   X := 2
```

Summary

Summary

- Some languages (Pascal) only support full evaluation of boolean expressions, some (Modula-2) only support short-circuit evaluation, others (Simula, Ada) allow the programmer to chose.
- Numeric representation is easier to implement, flow-of-control representation can be more efficient.
- We can often generate more efficient code by reversing tests ($< \Rightarrow \geq; \leq \Rightarrow >; \cdots$) to make the evaluation "fall through".

Summary...

- Predicting the outcome of tests (e.g. by feeding profiling information back into the compiler) is an important optimization technique.
- It is possible (but not always advisable) to use attribute grammars for (intermediate) code generation.
- Read the Dragon-book: 488–497, 468–469, 500-506