Control Structures

IF \( E \) THEN
\[ S \]
ENDIF

WHILE \( E \) DO
\[ S \]
ENDDO;

REPEAT
\[ S \]
UNTIL \( E \);
ENDLOOP;

LOOP
\[ S \]
ENDLOOP;

IF \( E \) THEN
\[ S \]
ELSE
\[ S \]
ENDIF;

IF \( E_1 \) THEN \( S_1 \)
ELSIF \( E_2 \) THEN \( S_2 \)
ELSE \( S_3 \)
ENDIF

FOR \( i: \) INT := \( E_1 \) TO \( E_2 \) BY \( E_3 \) DO
\[ S \]
ENDFOR;

FOR \( i \) IN \([E_1, E_2, \ldots]\) DO
\[ S \]
ENDFOR;

Boolean Expressions
Short Circuit Evaluation

With short circuit evaluation of boolean expressions we only evaluate as much of the expression as is necessary to determine if the expression evaluates 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) < 23) 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

- 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.
Boolean expressions are evaluated similarly to arithmetic expressions.

Operators:

- **FALSE** ≡ 0
- **TRUE** ≡ \(\text{any value} > 0\)
- \(x \text{ AND } y\) ≡ \(x \cdot y\)
- \(x \text{ OR } y\) ≡ \(x + y\)
- **NOT** \(x\) ≡ \(\text{IF } x = 0 \text{ THEN } t := 1 \text{ ELSE } t := 0;\)
- \(x < y\) ≡ \(\text{IF } x < y \text{ THEN } t := 1 \text{ ELSE } t := 0;\)

Flow-of-Control Representation

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 \(X<10=\text{TRUE}\)) or 101 (if \(X<10=\text{TRUE}\)).

\[
B := X < 10;
\]
\[
\text{WHILE } X > 5 \text{ DO }
\]
\[
\quad \text{DEC } (X);
\]
\[
\text{ENDDO}
\]

Flow-of-Control

100: **IF** \(X < 10\) **GOTO** 103
101: \(B := 0\)
102: **GOTO** 104
103: \(B := 1\)
104: **IF** \(X > 5\) **GOTO** 107
105: \(t_1 := 0\)
106: **GOTO** 108
107: \(t_1 := 1\)
108: **IF** \(t_1 = 0\) **GOTO** 111
109: \(X := X - 1;\)
110: **GOTO** 104
111:
Short-Circuit Evaluation

- 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

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

<table>
<thead>
<tr>
<th>Short Circuit</th>
<th>Full Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>100: IF X &lt;= 5 GOTO 103</td>
<td>100: $t_1 := X &gt; 5$;</td>
</tr>
<tr>
<td>101: IF NOT f(34) GOTO 103</td>
<td>101: $t_2 := f(34)$;</td>
</tr>
<tr>
<td>102: X := X - 1;</td>
<td>102: $t_3 := t_1 \text{ AND } t_2$;</td>
</tr>
<tr>
<td>103:</td>
<td>103: IF NOT $t_3$ GOTO 105</td>
</tr>
<tr>
<td>104: X := X - 1;</td>
<td>104:</td>
</tr>
<tr>
<td>105:</td>
<td>105:</td>
</tr>
</tbody>
</table>

Short-Circuit Code – OR

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

<table>
<thead>
<tr>
<th>Short Circuit</th>
<th>Full Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>100: IF X &gt; 5 GOTO 103</td>
<td>100: $t_1 := X &gt; 5$;</td>
</tr>
<tr>
<td>101: IF f(34) GOTO 103</td>
<td>101: $t_2 := f(34)$;</td>
</tr>
<tr>
<td>102: GOTO 104</td>
<td>102: $t_3 := t_1 \text{ OR } t_2$;</td>
</tr>
<tr>
<td>103: IF NOT $t_3$ GOTO 105</td>
<td>103:</td>
</tr>
<tr>
<td>104: X := X - 1;</td>
<td>104: X := X - 1;</td>
</tr>
<tr>
<td>105:</td>
<td>105:</td>
</tr>
</tbody>
</table>
Implementing 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.

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.

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
  IF a = 1 THEN
    X := 1;
  ENDIF
ENDDO
⇒
L1: if x >= b goto L2
  if a <> 1 goto L4
  X := 1
L4: goto L1
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' SS1 'ENDIF'

Example:

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

⇓

100:  IF X > 5 GOTO 102 (=E.true)
101:  GOTO 103 (=E.false)
102:  X := X - 1;
103:
The next rule \((S.\text{next} := S.\text{next};)\) 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 statement”.

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 \(E.\text{true}\), 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.

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

```
L1: X := 1
    if a < b then goto L2
goto L3
L2: X := 3;
L3: X := 2;
```
Control Structures – IF-ELSE

S ::= 'IF' E 'THEN' SS1 'ELSE' SS2 'ENDIF'

We have to generate two new labels, one attached 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 statement, hence we set S1.next and S2.next to S.next.

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 – **WHILE**

```plaintext
S ::= 'WHILE' E 'DO' SS1 'ENDDO'
```

- **S.begin** is a label we create and attach to the expression itself. Later we will complete the loop by generating a jump back to this label.
- **E.true** is a label we attach 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.

Example:

```plaintext
WHILE (X>5) DO DEC(X); ENDDO
```

```
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 ::= 'WHILE' E 'DO' SS1 'ENDDO'
S ::= E S1
{ S.begin := newlabel();
  E.true := newlabel();
  E.false := S.next;
  S1.next := S.begin;
  S.code := 'S.begin "":"' || E.code || 'E.true ":"' ||
            S1.code || 'goto S.begin'
}
```

- 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.
Relational Operators:

E ::= E1 < E2

E0 ::= E1 < E2

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

Not:

E ::= 'NOT' E1

E0 ::= NOT E1

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

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 E1 . true and E2 . true to the same label as E0 . true.

Example – OR

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

GOTO L3

Expr next
IF
  Stat
Lop Rop falsetrue
OR X := 1
Lop ... a < b GOTO L1
GOTO L2
IF c = d GOTO L1
IF a < b GOTO L1
GOTO L2
L2: IF c = d GOTO L1
GOTO L3
L1: X := 1;
L3:
Control Structures – \textbf{AND}

\[ E ::= E_1 \ 'AND' \ E_2 \]

\[ E_0 ::= E_1 \ AND \ E_2 \]
\{ \]
\begin{align*}
E_1\text{.true} & := \text{newlabel()}; \\
E_1\text{.false} & := E_0\text{.false}; \\
E_2\text{.true} & := E_0\text{.true}; \\
E_2\text{.false} & := E_0\text{.false}; \\
E\text{.code} & := E_1\text{.code} \parallel 'E_1\text{.true }':' \parallel E_2\text{.code}
\end{align*}
\}

\begin{tabular}{|c|c|}
\hline
\text{E\text{.code}} & \text{to E\text{.true}} \\hline
\text{E\text{.true}} & \text{to E\text{.false}=E\text{.false}} \\hline
\text{E\text{.false}} & \text{to E\text{.true}=E\text{.true}} \\hline
\text{E\text{.false}} & \text{to E\text{.false}=E\text{.false}} \\hline
\end{tabular}

\textbf{Example – AND}

\[
\begin{array}{l}
\text{IF (a < b) AND c = d THEN}
\end{array}
\]
\[
\begin{array}{c}
\begin{array}{c}
X := 1
\end{array}
\end{array}
\]
\[
\text{ENDIF}
\]

\begin{figure}
\centering
\includegraphics[width=\textwidth]{example.png}
\caption{Control Structure Diagram for AND}
\end{figure}

\textbf{Example – WHILE}

\[
\begin{array}{l}
\text{WHILE NOT (a > b) DO}
\end{array}
\]
\[
\begin{array}{c}
\begin{array}{c}
\text{IF c < d THEN X := 1 ENDIF}
\end{array}
\end{array}
\]
\[
\text{ENDDO}
\]

\begin{figure}
\centering
\includegraphics[width=\textwidth]{example.png}
\caption{Control Structure Diagram for WHILE}
\end{figure}

\textbf{Examples}

\begin{itemize}
\item IF a > b GOTO L3
\item GOTO L1
\item L1: IF c = d GOTO L4
\item GOTO L2
\item L4: X := 1;
\item GOTO L2
\item L3:
\end{itemize}
Example – IF-ELSE

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

GOTO L2

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 choose.

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”.

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