CS 4300: Compiler Theory

Chapter 6
Intermediate-Code Generation

Xuejun Liang

2019 Fall

Outlines (Sections)

1. Variants of Syntax Trees
2. Three-Address Code
3. Types and Declarations
4. Translation of Expressions
5. Type Checking
6. Control Flow
7. Backpatching
8. Switch-Statements
9. Intermediate Code for Procedures

6. Control Flow

Boolean Expressions

B → B || B | B && B | ! B | (B) | E **rel** E | **true** | **false**

rel represents: <, <=, =, ! =, >, or >=

Short-Circuit (or Jumping) Code Example

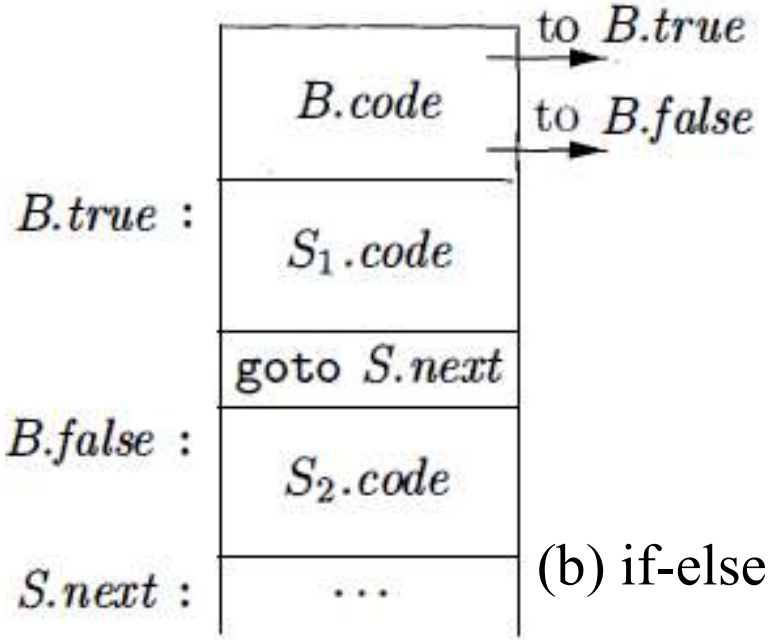
Statement: `if ( x < 100 || x > 200 && x != y ) x = 0;`

Jumping Code

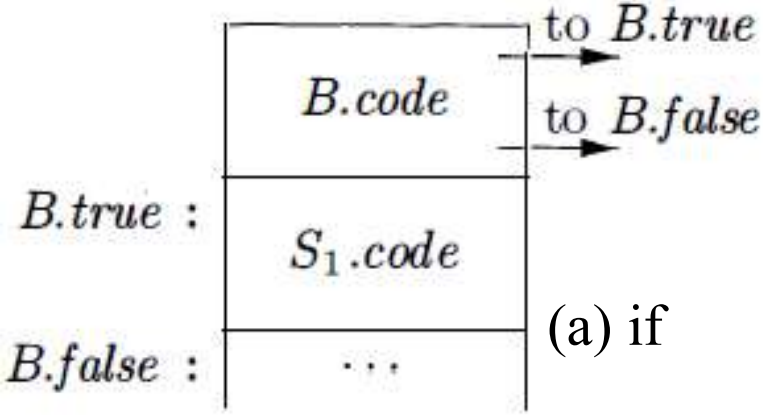
```
        if x < 100 goto L2
        if False x > 200 goto L1
        if False x != y goto L1
L2:     x = 0
L1:
```

Flow-of-Control Statements

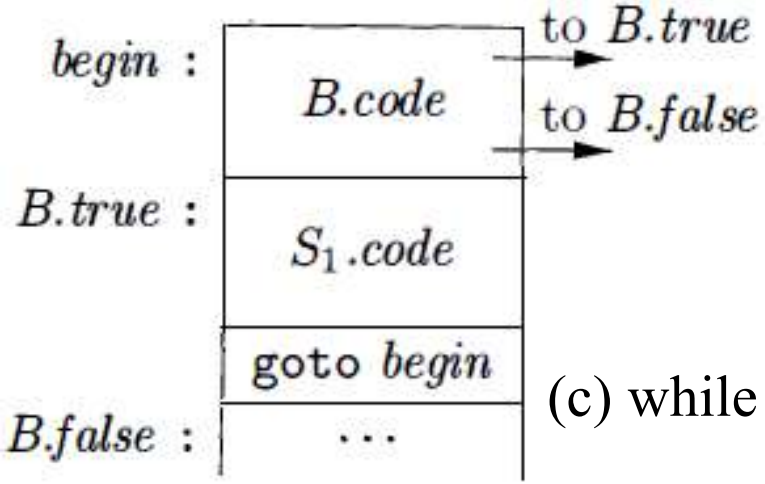
S	$\rightarrow$	**if** (B) S_1
S	$\rightarrow$	**if** (B) S_1 **else** S_2
S	$\rightarrow$	**while** (B) S_1



(b) if-else



(a) if



(c) while

Syntax-Directed Definition for Flow-of-Control Statements

PRODUCTION	SEMANTIC RULES
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \mid\mid label(S.next)$
$S \rightarrow$ **assign**	$S.code =$ **assign**$.code$
$S \rightarrow$ **if** (B) S_1	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \mid\mid label(B.true) \mid\mid S_1.code$
$S \rightarrow S_1\ S_2$	$S_1.next = newlabel()$ $S_2.next = S.next$ $S.code = S_1.code \mid\mid label(S_1.next) \mid\mid S_2.code$

Syntax-Directed Definition for Flow-of-Control Statements (Cont.)

$S \rightarrow$ **if** (B) S_1 **else** S_2	$B.true = newlabel()$ $B.false = newlabel()$ $S_1.next = S_2.next = S.next$ $S.code = B.code$ $\quad \| \| label(B.true) \| \| S_1.code$ $\quad \| \| gen('\texttt{goto}'\ S.next)$ $\quad \| \| label(B.false) \| \| S_2.code$
$S \rightarrow$ **while** (B) S_1	$begin = newlabel()$ $B.true = newlabel()$ $B.false = S.next$ $S_1.next = begin$ $S.code = label(begin) \| \| B.code$ $\quad \| \| label(B.true) \| \| S_1.code$ $\quad \| \| gen('\texttt{goto}'\ begin)$

PRODUCTION	SEMANTIC RULES
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$
$B \rightarrow B_1 \,\&\&\, B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.true) \mid\mid B_2.code$
$B \rightarrow ! B_1$	$B_1.true = B.false$ $B_1.false = B.true$ $B.code = B_1.code$
$B \rightarrow E_1 \textbf{ rel } E_2$	$B.code = E_1.code \mid\mid E_2.code$ $\mid\mid gen('\texttt{if}'\ E_1.addr\ \textbf{rel}.op\ E_2.addr\ '\texttt{goto}'\ B.true)$ $\mid\mid gen('\texttt{goto}'\ B.false)$
$B \rightarrow \textbf{true}$	$B.code = gen('\texttt{goto}'\ B.true)$
$B \rightarrow \textbf{false}$	$B.code = gen('\texttt{goto}'\ B.false)$

Control-Flow Translation of Boolean Expressions

Generating three-address code for Booleans

Examples

$B \rightarrow E_1$ **rel** E_2

```
a < b
```

```
if a < b goto B.true
goto B.false
```

```
if ( x < 100 || x > 200 && x != y )
    x = 0;
```

```
        if x < 100 goto L2
        goto L3
L3:     if x > 200 goto L4
        goto L1
L4:     if x != y goto L2
        goto L1
L2:     x = 0
L1:
```

Avoiding Redundant Gotos

$S \rightarrow \textbf{if}\ (B)\ S_1$

$$
\begin{aligned}
B.true &= fall \\
B.false &= S_1.next = S.next \\
S.code &= B.code \,||\, S_1.code
\end{aligned}
$$

$B \rightarrow B_1\ ||\ B_2$

$$
\begin{aligned}
B_1.true &= \textbf{if}\ B.true \neq fall\ \textbf{then}\ B.true\ \textbf{else}\ newlabel() \\
B_1.false &= fall \\
B_2.true &= B.true \\
B_2.false &= B.false \\
B.code &= \textbf{if}\ B.true \neq fall\ \textbf{then}\ B_1.code \,||\, B_2.code \\
&\quad\ \textbf{else}\ B_1.code \,||\, B_2.code \,||\, label(B_1.true)
\end{aligned}
$$

$B \rightarrow E_1\ \textbf{rel}\ E_2$

$$
\begin{aligned}
test &= E_1.addr\ \textbf{rel}.op\ E_2.addr \\
s &= \textbf{if}\ B.true \neq fall\ \textbf{and}\ B.false \neq fall\ \textbf{then} \\
&\quad\ gen(\texttt{'if'}\ test\ \texttt{'goto'}\ B.true) \,||\, gen(\texttt{'goto'}\ B.false) \\
&\quad\ \textbf{else if}\ B.true \neq fall\ \textbf{then}\ gen(\texttt{'if'}\ test\ \texttt{'goto'}\ B.true) \\
&\quad\ \textbf{else if}\ B.false \neq fall\ \textbf{then}\ gen(\texttt{'ifFalse'}\ test\ \texttt{'goto'}\ B.false) \\
&\quad\ \textbf{else}\ \texttt{''} \\
B.code &= E_1.code \,||\, E_2.code \,||\, s
\end{aligned}
$$

7. Backpatching

Backpatching can be used to generate code for Boolean expressions and flow-of-control statements in one pass

Three synthesized attributes, each with a list of jumps ⇒

B.truelist
B.falselist
S.nextlist

Three functions to manipulate lists of jumps ⇒

makelist(*i*)
merge(P_1, P_2)
backpatch(*p*, *i*)

Note: Generated instructions are stored into an instruction array, and thus labels will be indices into this array.

Backpatching for Boolean Expressions

$B \rightarrow B_1 \,||\, M\ B_2 \mid B_1\ \&\&\ M\ B_2 \mid !\ B_1 \mid (B_1) \mid E_1$ **rel** $E_2 \mid$ **true** $\mid$ **false**
$M \rightarrow \varepsilon$

1) $B \rightarrow B_1 \,||\, M\ B_2$ { $backpatch(B_1.falselist, M.instr)$;
$B.truelist = merge(B_1.truelist, B_2.truelist)$;
$B.falselist = B_2.falselist$; }

2) $B \rightarrow B_1\ \&\&\ M\ B_2$ { $backpatch(B_1.truelist, M.instr)$;
$B.truelist = B_2.truelist$;
$B.falselist = merge(B_1.falselist, B_2.falselist)$; }

3) $B \rightarrow\ !\ B_1$ { $B.truelist = B_1.falselist$;
$B.falselist = B_1.truelist$; }

4) $B \rightarrow (\ B_1\ )$ { $B.truelist = B_1.truelist$;
$B.falselist\ =\ B_1.falselist$; }

Translation scheme for Boolean expressions

Backpatching for Boolean Expressions

$B \rightarrow B_1 \,||\, M\ B_2 \mid B_1\ \&\&\ M\ B_2 \mid !\ B_1 \mid (B_1) \mid E_1$ **rel** $E_2 \mid$ **true** $\mid$ **false**
$M \rightarrow \varepsilon$

5) $B \rightarrow E_1$ **rel** E_2 { $B.truelist = makelist(nextinstr)$;
$B.falselist = makelist(nextinstr + 1)$;
$emit('\texttt{if}'\ E_1.addr\ \textbf{rel}.op\ E_2.addr\ '\texttt{goto}\ \_')$;
$emit('\texttt{goto}\ \_')$; }

6) $B \rightarrow$ **true** { $B.truelist = makelist(nextinstr)$;
$emit('\texttt{goto}\ \_')$; }

7) $B \rightarrow$ **false** { $B.falselist = makelist(nextinstr)$;
$emit('\texttt{goto}\ \_')$; }

8) $M \rightarrow \epsilon$ { $M.instr = nextinstr$; }

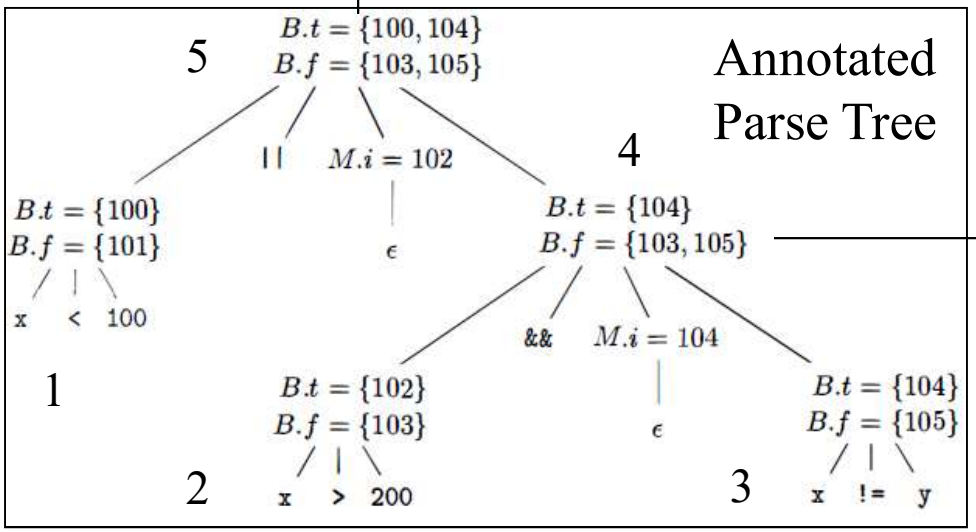
Translation scheme for Boolean expressions

Example 6.24

x < 100 || x > 200 && x != y

5

```
100:  if x < 100 goto _
101:  goto 102
102:  if y > 200 goto 104
103:  goto _
104:  if x != y goto _
105:  goto _
```



Annotated Parse Tree

1

```
100: if x < 100 goto _
101: goto _
```

2

```
102: if x > 200 goto _
103: goto _
```

3

```
104: if x ! = y goto _
105: goto _
```

4

```
100:  if x < 100 goto _
101:  goto _
102:  if x > 200 goto 104
103:  goto _
104:  if x != y goto _
105:  goto _
```

Flow-of- Control Statements

$S \rightarrow$ **if** (B) S | **if** (B) S else S | **while** (B) S | { L } | A ;
$L \rightarrow L\ S \mid S$

1) $S \rightarrow$ **if** (B) $M\ S_1$ { *backpatch*($B.truelist$, $M.instr$);
S.nextlist = *merge*($B.falselist$, $S_1.nextlist$); }

2) $S \rightarrow$ **if** (B) $M_1\ S_1\ N$ **else** $M_2\ S_2$
{ *backpatch*($B.truelist$, $M_1.instr$);
backpatch($B.falselist$, $M_2.instr$);
temp = *merge*($S_1.nextlist$, $N.nextlist$);
S.nextlist = *merge*(*temp*, $S_2.nextlist$); }

3) $S \rightarrow$ **while** M_1 (B) $M_2\ S_1$
{ *backpatch*($S_1.nextlist$, $M_1.instr$);
backpatch($B.truelist$, $M_2.instr$);
S.nextlist = $B.falselist$;
emit('goto' $M_1.instr$); }

Flow-of- Control Statements (Cont.)

$S \rightarrow$ **if** (B) S | **if** (B) S else S | **while** (B) S | { L } | A ;
$L \rightarrow L\ S \mid S$

4) $S \rightarrow \{\ L\ \}$ { $S.nextlist = L.nextlist$; }

5) $S \rightarrow A\ ;$ { $S.nextlist =$ **null**; }

6) $M \rightarrow \epsilon$ { $M.instr = nextinstr$; }

7) $N \rightarrow \epsilon$ { $N.nextlist = makelist(nextinstr)$;
$emit('\texttt{goto}\ \_')$; }

8) $L \rightarrow L_1\ M\ S$ { $backpatch(L_1.nextlist,\ M.instr)$;
$L.nextlist = S.nextlist$; }

9) $L \rightarrow S$ { $L.nextlist = S.nextlist$; }

Exercise 6.7.2

```
while (E1) {
    if (E2)
        while (E3)
            S1
    else {
        if (E4)
            S2;
        S3;
    }
}
```

Control-flow

i_1: Code for E_1
i_2: Code for E_2
i_3: Code for E_3
i_4: Code for S_1
i_5: Code for E_4
i_6: Code for S_2
i_7: Code for S_3
i_8: ...

Three-address

(a) E_3.false = i_1
(b) S_2.next = i_7
(c) E_4.false = i_7
(d) S_1.next = i_3
(e) E_2.true = i_3

8. Switch- Statements

Translation of a switch-statement

```
        code to evaluate E into t
        goto test
L1:     code for S1
        goto next
L2:     code for S2
        goto next
        ...
Ln-1:   code for Sn-1
        goto next
Ln:     code for Sn
        goto next
test:   if t = V1 goto L1
        if t = V2 goto L2
        ...
        if t = Vn-1 goto Ln-1
        goto Ln
next:
```

Switch-statement

```
switch ( E ) {
        case V1: S1
        case V2: S2
                 ...
        case Vn-1: Sn-1
        default: Sn
}
```

Case three-address-code instructions

```
case t V1 L1
case t V2 L2
...
case t Vn-1 Ln-1
case t t Ln
label next
```

9. Intermediate Code fo r Procedures

n = f(a[i]);

1) t_1 = i * 4
2) t_2 = a [t_1]
3) param t_2
4) t_3 = call f, 1
5) n = t_3

Function definitions and function calls

$$
\begin{array}{lcl}
D & \rightarrow & \textbf{define}\ T\ \textbf{id}\ (\ F\ )\ \{\ S\ \} \\
F & \rightarrow & \epsilon \mid T\ \textbf{id}\ ,\ F \\
S & \rightarrow & \textbf{return}\ E\ ; \\
E & \rightarrow & \textbf{id}\ (\ A\ ) \\
A & \rightarrow & \epsilon \mid E\ ,\ A
\end{array}
$$

Four concepts used for translation of function definitions and function calls

Function types
Symbol tables
Type checking
Function calls