# Compiler Design

Fatemeh Deldar

Isfahan University of Technology

1403-1404

# Translation of Expressions

- **Goal**
  - Translation of expressions and statements into three-address code
- **Example**
  - Attributes ***S.code*** and ***E.code*** denote the three-address code for ***S*** and ***E***, respectively
  - Attribute ***E.addr*** denotes the address that will hold the value of ***E***
    - **Recall that an address can be a name, a constant, or a compiler-generated temporary**

| PRODUCTION | SEMANTIC RULES |
|---|---|
| $S \rightarrow \mathbf{id} = E \,;$ | $S.code = E.code \;\Vert$ <br> $gen(top.get(\mathbf{id}.lexeme)\ '='\ E.addr)$ |
| $E \rightarrow E_1 + E_2$ | $E.addr = \mathbf{new}\ Temp\,()$ <br> $E.code = E_1.code \;\Vert\; E_2.code \;\Vert$ <br> $gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$ |
| $\mid\ -\ E_1$ | $E.addr = \mathbf{new}\ Temp\,()$ <br> $E.code = E_1.code \;\Vert$ <br> $gen(E.addr\ '='\ '\mathbf{minus}'\ E_1.addr)$ |
| $\mid\ (\ E_1\ )$ | $E.addr = E_1.addr$ <br> $E.code = E_1.code$ |
| $\mid\ \mathbf{id}$ | $E.addr = top.get(\mathbf{id}.lexeme)$ <br> $E.code = ''$ |

**Let *top* denote the current symbol table.**

# Translation of Expressions

- **Example**
  - The SDD in the previous slide translates the assignment statement $\boldsymbol{a} = \boldsymbol{b} + -\boldsymbol{c}$; into the following three-address code sequence

$$
\begin{array}{l}
\texttt{t}_1 \ \texttt{= minus c} \\
\texttt{t}_2 \ \texttt{= b + t}_1 \\
\texttt{a = t}_2
\end{array}
$$

# Incremental Translation

- Code attributes can be long strings, so they are usually generated incrementally
- In the incremental approach, gen not only constructs a three-address instruction, it appends the instruction to the sequence of instructions generated so far

$$
\begin{array}{lcll}
S & \rightarrow & \textbf{id} = E \; ; & \{ \; gen(\; top.get(\textbf{id}.lexeme) \; '=' \; E.addr); \; \} \\[2ex]
E & \rightarrow & E_1 + E_2 & \{ \; E.addr = \textbf{new} \; Temp\,(); \\
 & & & \quad gen(E.addr \; '=' \; E_1.addr \; '+' \; E_2.addr); \; \} \\[2ex]
 & | & - \; E_1 & \{ \; E.addr = \textbf{new} \; Temp\,(); \\
 & & & \quad gen(E.addr \; '=' \; '\textbf{minus}' \; E_1.addr); \; \} \\[2ex]
 & | & ( \; E_1 \; ) & \{ \; E.addr = E_1.addr; \; \} \\[2ex]
 & | & \textbf{id} & \{ \; E.addr = top.get(\textbf{id}.lexeme); \; \}
\end{array}
$$

# Type Conversions

- **Example**
  - Suppose that integers are converted to floats when necessary, using a unary operator (float)

$$2 * 3.14 \quad \Longrightarrow \quad \begin{array}{l} t_1 = \texttt{(float) } 2 \\ t_2 = t_1 * 3.14 \end{array}$$

- Type conversion rules vary from language to language
- Conversion from one type to another is said to be implicit if it is done automatically by the compiler
- Conversion is said to be explicit if the programmer must write something to cause the conversion

# Type Conversions

- **Example:** Introducing type conversions into expression evaluation

$$\boxed{E \rightarrow E_1 + E_2} \quad \{ \begin{array}{l} E.type \ = \ max(E_1.type, E_2.type); \\ a_1 \ = \ widen(E_1.addr, E_1.type, E.type); \\ a_2 \ = \ widen(E_2.addr, E_2.type, E.type); \\ E.addr = \textbf{new} \ Temp\,(); \\ gen(E.addr \ '=' \ a_1 \ '+' \ a_2); \ \} \end{array}$$

  - $\boldsymbol{max(t_1, t_2)}$ takes two types $t_1$ and $t_2$ and returns the maximum
  - $\boldsymbol{widen(a, t, w)}$ generates type conversions if needed to widen the contents of an address $a$ of type $t$ into a value of type $w$

# Control Flow

- The translation of statements such as if-else-statements and while-statements is tied to the translation of boolean expressions

- Boolean expressions are composed of the boolean operators (which we denote &&, | |, and !, using the C convention for the operators AND, OR, and NOT, respectively) applied to elements that are boolean variables or relational expressions

- **Example** $\boxed{B \rightarrow B \;||\; B \mid B \;\&\&\; B \mid \;!\; B \mid (\; B\; ) \mid E \;\textbf{rel}\; E \mid \textbf{true} \mid \textbf{false}}$
  - We use the attribute $rel.op$ to indicate which of the six comparison operators <, <=, =, ! =, >, or >= is represented by $rel$

- Given the expression $B_1||B_2$, if we determine that $B_1$ is true, then we can conclude that the entire expression is true without having to evaluate $B_2$. Similarly, given $B_1\&\&B_2$, if $B_1$ is false, then the entire expression is false.

# Control Flow

- **Short-Circuit Code**
  - In short-circuit (or jumping) code, the boolean operators &&, ||, and ! translate into jumps
  - **Example**

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

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

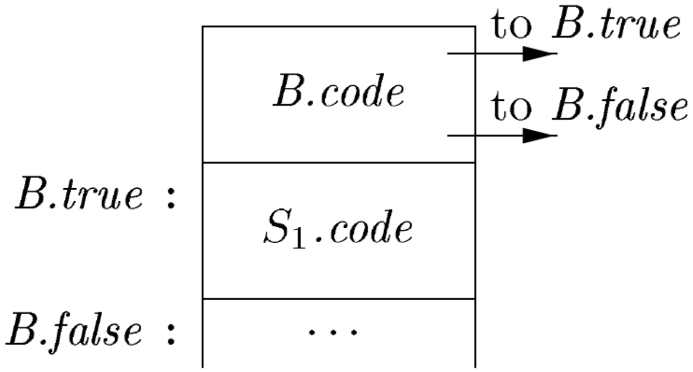
# Control Flow

- **Flow-of-Control Statements**
  - **Example**

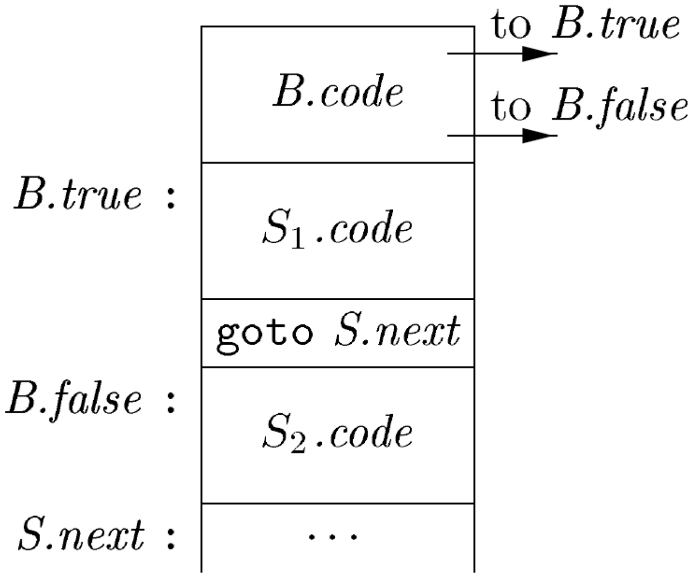
$$
\begin{array}{lcl}
S & \rightarrow & \textbf{if} \ ( \ B \ ) \ S_1 \\
S & \rightarrow & \textbf{if} \ ( \ B \ ) \ S_1 \ \textbf{else} \ S_2 \\
S & \rightarrow & \textbf{while} \ ( \ B \ ) \ S_1
\end{array}
$$

    - Non-terminal ***B*** represents a boolean expression and non-terminal ***S*** represents a statement

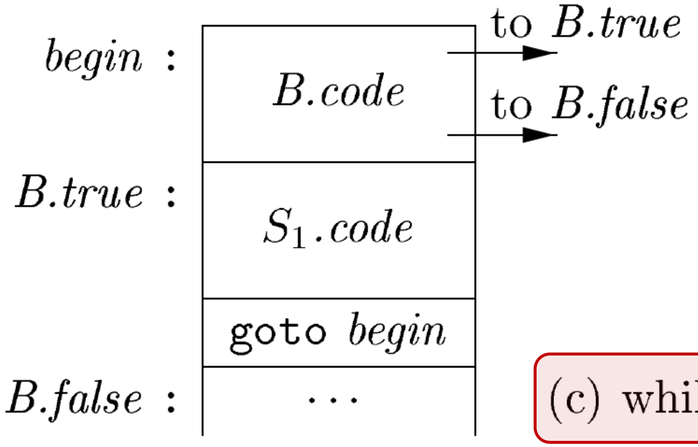
# Control Flow




(a) if




(b) if-else




(c) while