Code Generation

- Intermediate code generation: Abstract (machine independent) code.
- Code optimization: Transformations to the code to improve time/space performance.
- Final code generation: Emitting machine instructions.

Syntax Directed Translation

Interpretation:

$E \longrightarrow E_1 + E_2 \quad \{\ E.val := E_1.val + E_2.val;\ \}$

Type Checking:

$E \longrightarrow E_1 + E_2 \quad \{$
if $E_1.type \equiv E_2.type \equiv int$
$E.type = int;$
else
$E.type = float;$
$\}$

Code Generation via Syntax Directed Translation

Code Generation:

$$
\begin{array}{lll}
E & \longrightarrow & E_1 \texttt{ + } E_2 \quad \{ \\
 & & E.code = E_1.code \parallel \\
 & & \qquad E_2.code \parallel \\
 & & \qquad emit(\texttt{add}) \\
 & & \}
\end{array}
$$

Code Genetation and Attributes

$$
\begin{array}{lll}
E & \longrightarrow & E_1 \texttt{ + } E_2 \quad \{ \\
 & & E.code = E_1.code \parallel \\
 & & \qquad E_2.code \parallel \\
 & & \qquad emit(\texttt{add}) \\
 & & \} \\
E & \longrightarrow & \texttt{int} \quad \{ \\
 & & E.code = emit(\texttt{load\_immed}, \texttt{int}.val) \\
 & & \} \\
E & \longrightarrow & \texttt{id} \quad \{ \\
 & & E.code = emit(\texttt{load}, \texttt{int}.addr) \\
 & & \}
\end{array}
$$

Generating 3-address code

$$
\begin{array}{lcll}
E & \longrightarrow & E_1 \texttt{ + } E_2 & \{ \\
 & & & E.temp = generate\_new\_temporary(); \\
 & & & E.code = E_1.code \parallel \\
 & & & \qquad E_2.code \parallel \\
 & & & \qquad emit(\texttt{add } E_1.temp,\ E_2.temp,\ E.temp\ ); \\
 & & & \} \\
E & \longrightarrow & \texttt{int} & \{ \\
 & & & E.temp = generate\_new\_temporary(); \\
 & & & E.code = emit(\texttt{mov\_immed int}.val,\ E.temp); \\
 & & & \} \\
E & \longrightarrow & \texttt{id} & \{ \\
 & & & E.temp = \texttt{id}.addr; \\
 & & & E.code = ''; \\
 & & & \}
\end{array}
$$

l- and *r*-Values

```
i := i + 1;
```

- *l*-**value**: location where the value of the expression is stored.
- *r*-**value**: actual value of the expression

Computing *l*-values

$$
\begin{array}{lll}
E & \longrightarrow & \texttt{id} \\
 & & \{ \\
 & & E.lval = generate\_new\_temporary(); \\
 & & E.lcode = emit(\texttt{mov\_immed id}.addr,\ E.lval) \\
 & & \} \\
E & \longrightarrow & E_1\ \texttt{[}\ E_2\ \texttt{]} \\
 & & \{ \\
 & & E.lval = generate\_new\_temporary(); \\
 & & E.lcode = E_1.lcode \parallel \\
 & & \qquad E_2.code \parallel \\
 & & \qquad emit(\texttt{add}\ E_1.lval,\ E_2.temp,\ E.lval\ ) \\
 & & \} \\
E & \longrightarrow & E_1\ \texttt{.}\ \texttt{id} \\
 & & \{ \\
 & & E.lval = generate\_new\_temporary(); \\
 & & E.lcode = E_1.lcode \parallel \\
 & & \qquad emit(\texttt{add}\ E_1.lval,\ \texttt{id}.offset,\ E.lval\ ) \\
 & & \}
\end{array}
$$

Code Generation for Assignment

$$
\begin{array}{lll}
E & \longrightarrow & E_1\ \texttt{=}\ E_2 \\
 & & \{ \\
 & & E.code = E_1.lcode \parallel \\
 & & \qquad E_2.code \parallel \\
 & & \qquad emit(\texttt{move}\ E_2.temp,\ E_1.lval) \\
 & & E.temp = E_2.temp \\
 & & \}
\end{array}
$$

Code Generation for Other Expressions

$$
\begin{array}{lll}
E & \longrightarrow & E_1 \texttt{ [} E_2 \texttt{]} \quad \{ \\
 & & \quad E.lval = generate\_new\_temporary(); \\
 & & \quad E.temp = generate\_new\_temporary(); \\
 & & \quad E.lcode = E_1.code \parallel \\
 & & \qquad\qquad E_2.code \parallel \\
 & & \qquad\qquad emit(\texttt{add } E_1.temp, E_2.temp, E.lval\ ) \\
 & & \quad E.code = E.lcode \parallel \\
 & & \qquad\qquad emit(\texttt{move } E.lval, E.temp) \\
 & & \}
\end{array}
$$

Function Calls (Method Invocations)

Using Call-by-value.

$$
\begin{array}{lll}
E & \longrightarrow & \texttt{id (} E_1, E_2 \texttt{)} \quad \{ \\
 & & \quad E.temp = generate\_new\_temporary(); \\
 & & \quad E.code = E_1.code \parallel \\
 & & \qquad\qquad E_2.code \parallel \\
 & & \qquad\qquad emit(\texttt{push } E_1.temp) \\
 & & \qquad\qquad emit(\texttt{push } E_2.temp) \\
 & & \qquad\qquad emit(\texttt{call id}.addr) \\
 & & \qquad\qquad emit(\texttt{pop } E.temp) \\
 & & \}
\end{array}
$$

Function Calls (Method Invocations)

Using Call-by-reference.

$$
\begin{array}{lll}
E & \longrightarrow & \texttt{id (} E_1\texttt{, } E_2\texttt{)} \quad \{ \\
 & & E.temp = generate\_new\_temporary(); \\
 & & E.code = E_1.lcode \parallel \\
 & & \qquad E_2.lcode \parallel \\
 & & \qquad emit(\texttt{push } E_1.lval) \\
 & & \qquad emit(\texttt{push } E_2.lval) \\
 & & \qquad emit(\texttt{call id}.addr) \\
 & & \qquad emit(\texttt{pop } E.temp) \\
 & & \}
\end{array}
$$

Code Generation for Statements

$$
\begin{array}{lll}
S & \longrightarrow & S_1\texttt{ ; }S_2 \quad \{ \\
 & & S.code = S_1.code \parallel \\
 & & \qquad S_2.code; \\
 & & \} \\
S & \longrightarrow & E \quad \{ \\
 & & S.code = E.code; \\
 & & \}
\end{array}
$$

Conditional Statements

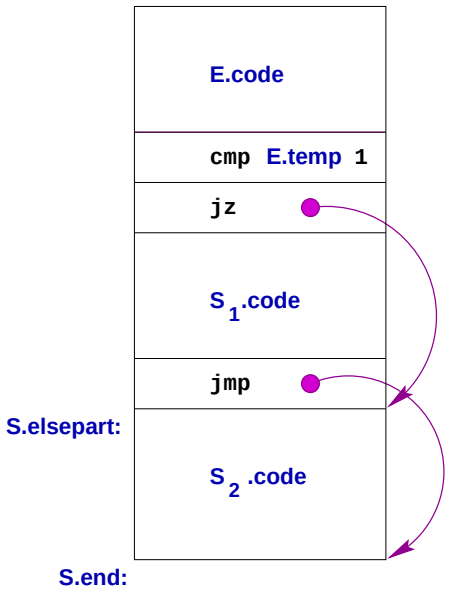
$$
\begin{array}{lll}
S & \longrightarrow & \texttt{if}\ E, S_1, S_2 \quad \{ \\
& & \quad elselabel = get\_new\_label(); \\
& & \quad endlabel = get\_new\_label(); \\
& & \quad S.code = \quad E.code \parallel \\
& & \qquad\qquad emit(\texttt{cmp}\ E.temp, \texttt{0}) \\
& & \qquad\qquad emit(\texttt{jz}\ elselabel) \\
& & \qquad\qquad S_1.code; \\
& & \qquad\qquad emit(\texttt{jmp}\ endlabel) \\
& & \qquad\quad emit(elselabel:) \\
& & \qquad\qquad S_2.code; \\
& & \qquad\quad emit(endlabel:) \\
& & \}
\end{array}
$$

Conditional Statements with Attributes

$$
\begin{array}{lll}
S & \longrightarrow & \texttt{if}\ E, S_1, S_2 \quad \{ \\
& & \quad S.begin = get\_new\_label(); \\
& & \quad S_1.end = get\_new\_label(); \\
& & \quad S_2.end = S.end; \\
& & \quad S.code = emit(S.begin:) \\
& & \qquad\qquad E.code \parallel \\
& & \qquad\qquad emit(\texttt{cmp}\ E.temp, \texttt{0}) \\
& & \qquad\qquad emit(\texttt{jz}\ S_2.begin) \\
& & \qquad\qquad S_1.code; \\
& & \qquad\quad emit(S_1.end:) \\
& & \qquad\qquad emit(\texttt{jmp}\ S.end) \\
& & \qquad\qquad S_2.code; \\
& & \}
\end{array}
$$

Code Generation for Statements

$S \longrightarrow$ `if` E, S_1, S_2



Conditional Statements

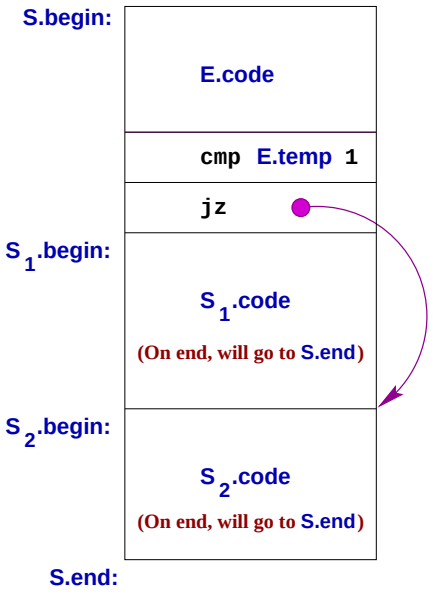
$S \longrightarrow$ `if` E, S_1, S_2 {

$elselabel = get\_new\_label();$
$endlabel = get\_new\_label();$
$S.code = E.code \parallel$
$emit(\texttt{cmp}\ E.temp, \texttt{1}) \parallel$
$emit(\texttt{jz}\ elselabel) \parallel$
$S_1.code \parallel$
$emit(\texttt{jmp}\ endlabel) \parallel$
$emit(elselabel\texttt{:}) \parallel$
$S_2.code \parallel$
$emit(endlabel\texttt{:})$

}

If Statements: An Alternative

$S \longrightarrow$ `if` E, S_1, S_2



Conditional Statements and Continuations

$S \longrightarrow$ `if` E, S_1, S_2 $\{$

$S.begin = get\_new\_label();$
$S_1.end = S_2.end = S.end;$
$S.code = emit(S.begin:) \parallel$
$E.code \parallel$
$emit(\texttt{cmp}\ E.temp, 1) \parallel$
$emit(\texttt{jz}\ S_2.begin) \parallel$
$S_1.code \parallel$
$S_2.code;$

$\}$

Continuations

An attribute of a statement that specifies where control will flow to **after** the statement is executed.

- Analogous to the *follow* sets of grammar symbols.
- In deterministic languages, there is only one continuation for each statement.
- Can be generalized to include local variables whose values are needed to execute the following statements:

  Uniformly captures call, return *and* **exceptions**.

Code Generation for Boolean Expressions

$$
\begin{array}{lcll}
E & \longrightarrow & E_1 \texttt{ \&\& } E_2 & \{ \\
 & & & E.code = E_1.code \parallel \\
 & & & \qquad E_2.code \parallel \\
 & & & \qquad emit(\texttt{and}) \\
 & & & \} \\
E & \longrightarrow & \texttt{! } E_1 & \{ \\
 & & & E.code = E_1.code \parallel \\
 & & & \qquad emit(\texttt{not}) \\
 & & & \} \\
E & \longrightarrow & \texttt{true} & \{ \\
 & & & E.code = emit(\texttt{load\_immed}, \texttt{1}) \\
 & & & \} \\
E & \longrightarrow & \texttt{id} & \{ \\
 & & & E.code = emit(\texttt{load}, \texttt{id}.addr) \\
 & & & \}
\end{array}
$$

Code for Boolean Expressions

```
if ((p != NULL)                    load(p);
  && (p->next != q)) {             null();
  ... then part                    aneq();
} else {                           load(p);
  ... else part                    ildc(1);
}                                  getfield();
                                   load(q);
                                   aneq();
                                   and();
                                   jnz elselabel;
                                   ...    then part
                                 elselabel:
                                   ...    else part
```


Shortcircuit Code

```
if ((p != NULL)                    load(p);
  && (p->next != q)) {             null();
  ... then part                    aneq();
} else {                           jnz elselabel;
  ... else part                    load(p);
}                                  ildc(1);
                                   getfield();
                                   load(q);
                                   aneq();
                                   jnz elselabel;
                                   ...    then part
                                 elselabel:
                                   ...    else part
```

Generating Shortcircuit Code

Use two continuations for each boolean expression:

- $E.success$: where control will go when expression in E evaluates to *true*.
- $E.fail$: where control will go when expression in E evaluates to *false*.

Shortcircuit Code for Boolean Expressions

$$
\begin{array}{lll}
E \longrightarrow E_1 \texttt{ \&\& } E_2 & \{ \\
& E_1.fail = E.fail; \\
& E_2.fail = E.fail; \\
& E_1.success = get\_new\_label(); \\
& E_2.success = E.success; \\
& E.code = E_1.code \parallel \\
& \qquad E_2.code \\
& \} \\
E \longrightarrow \texttt{! } E_1 & \{ \\
& E_1.fail = E.success; \\
& E_1.success = E.fail; \\
& \} \\
E \longrightarrow \texttt{true} & \{ \\
& E.code = emit(\texttt{jmp}, E.success) \\
& \}
\end{array}
$$

Short-circuit code for Conditional Statements

$$
\begin{array}{lll}
S & \longrightarrow & \texttt{if}\ E,\ S_1,\ S_2 \quad \{ \\
& & \quad S.begin = get\_new\_label(); \\
& & \quad S_1.end = S_2.end = S.end; \\
& & \quad E.success = S_1.begin; \\
& & \quad E.fail = S_2.begin; \\
& & \quad S.code = emit(S.begin:) \parallel \\
& & \qquad\qquad E.code \parallel \\
& & \qquad\qquad S_1.code \parallel \\
& & \qquad\qquad S_2.code; \\
& & \} \\
\end{array}
$$

Continuations and Code Generation

Continuation of a statement is an inherited attribute.

It is not an L-inherited attribute!

Code of statement is a synthesized attribute, but is dependent on its continuation.

Backpatching: *Make two passes to generate code.*

1. Generate code, leaving "holes" where continuation values are needed.
2. Fill these holes on the next pass.

What's left?

After intermediate code is generated,

- **Optimize** intermediate code using target machine-independent techniques.
  Examples:
  - constant propagation
  - loop-invariant code motion
  - dead-code elimination
  - strength reduction
- **Generate** final machine code
  Perform target machine-specific optimizations.