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

Coercion

Interpretation:

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

Type Checking:

$$E \longrightarrow E_1 + E_2 \quad \{$$

$$\text{if } E_1.type \equiv E_2.type \equiv int$$

$$E.type = int;$$

$$\text{else}$$

$$E.type = float;$$

$$\}$$

Code Generation via Syntax Directed Translation

Print 3+(2*5)

Code Generation:

$$E \longrightarrow E_1 + E_2 \qquad \{ \; E.code = E_1.code \,||\, E_2.code \,||\, \text{“add”}) \; \}$$

Postfix code

Intermediate Code

"Abstract" code generated from AST

- **Simplicity and Portability**
  - Machine independent code.
  - Enables common optimizations on intermediate code.
  - Machine-dependent code optimizations postponed to last phase.

Intermediate Forms

- *Stack machine code:*

  Code for a "postfix" stack machine.

- *Two address code:*

  Code of the form "add r_1, r_2"

  $r_2 = r_1 + r_2$

- *Three address code:*

  Code of the form "add $src_1, src_2, dest$"

  dest = src1 + src2

  Quadruples and Triples: Representations for three-address code.

Quadruples

Explicit representation of three-address code.

Eixample: `a := a + b * -c;`

Instr	Operation	Arg 1	Arg 2	Result
(0)	`uminus`	`c`		t_1
(1)	`mult`	`b`	t_1	t_2
(2)	`add`	`a`	t_2	t_3
(3)	`move`	t_3		`a`

Triples

Representation of three-address code with implicit destination argument.

Example: `a := a + b * -c;`

Instr	Operation	Arg 1	Arg 2
(0)	`uminus`	`c`	
(1)	`mult`	`b`	(0)
(2)	`add`	`a`	(1)
(3)	`move`	`a`	(2)

Intermediate Forms

Choice depends on convenience of further processing

- Stack code is simplest to generate for expressions.
- Quadruples are most general, permitting most optimizations including code motion.
- Triples permit optimizations such as *common subexpression elimination*, but code motion is difficult.

Static Single Assignment (SSA)

- Each variable is assigned at most once
- ϕ nodes used to combine values of variables after a conditional

```
if (f) x = 1; else x=2;
y=x*x;
```

Becomes

```
if (f) x1 = 1; else x2=2;
x3 = φ(x1, x2);
y=x3*x3;
```

merging

LLVM

$x = 6$ $x = 3$

x

$x_1 = 3;$
if (c)
$x_2 = 5$
$x_3 = \phi(x_1, x_2)$

Generating 3-address code

$E \longrightarrow E_1 + E_2$ {
  $E.addr$ = $newtemp()$;
  $E.code$ = $E_1.code \,||\, E_2.code \,||$
    $E.addr \,||\, ':=' \,||\, E_1.addr \,||\, '+' \,||\, E_2.addr$;
}

$E \longrightarrow$ int {
  $E.addr$ = $newtemp()$;
  $E.code$ = $E.addr \,||\, ':=' \,||\,$ int$.val$;
}

$E \longrightarrow$ id {
  $E.addr$ = id$.name$;
  $E.code$ = ''; // no code needed
}

x0
x1
x2

string concatenation

E.code is a string-valued attribute containing the code for E.

It generates a temp var for storing the result of evaluating E, & generates code to evaluate E and store in this temp.

Generation of Postfix Code for Boolean Expressions

$$
\begin{aligned}
E \quad & \longrightarrow E_1 \;\&\&\; E_2 \;\{ \\
& \quad E.code = E_1.code \;||\; \\
& \qquad\qquad\quad E_2.code \;||\; \\
& \qquad\qquad\quad gen(\mathtt{and}) \\
& \}
\end{aligned}
$$

$$
\begin{aligned}
E \quad & \longrightarrow \;!\; E_1 \;\{ \\
& \quad E.code = E_1.code \;||\; \\
& \qquad\qquad\quad gen(\mathtt{not}) \\
& \}
\end{aligned}
$$

$$E \quad \longrightarrow \mathrm{true} \;\{E.code = gen(\mathtt{load\_immed}, 1)\}$$

$$E \quad \longrightarrow \mathrm{id} \;\{E.code = gen(\mathtt{load}, \mathrm{id}.addr)\}$$

Code for Boolean Expressions

```
if ((p != NULL)
   && (p->next != q)) {
   ... then part
} else {
   ... else part
}
```

E_1 E_2

E's code

```
load(p);
null();
neq();
load(p);
ildc(1);
 getfield();
load(q);
neq();
and();
 jnz elselabel;
 ...      then part
elselabel:
 ...      else part
```

E_1 code

E_2 code

Shortcircuit Code

```
if ((p != NULL)
   && (p->next != q)) {
   ... then part
} else {
   ... else part
}
```

```
load(p);
null();
neq();
 jnz elselabel;
load(p);
ildc(1);
 getfield();
load(q);
neq();
 jnz elselabel;
 ...     then part
elselabel:
 ...     else part
```

l- and *r*-Values

l-value   r-value   a[i]

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

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

  – for variables, the r-value is the value stored at the variable's location (which, in turn, is the l-value of the variable)

Computing *l*-values

$E \longrightarrow$ id {

$E.lval$ = id.loc;

$E.code$ = ‘’; }

// empty code

loc attribute stores location of the variable id.

$E \longrightarrow E_1$ [E_2] {

$E.lval = newtemp();$

$x = newtemp();$

$E.lcode = E_1.lcode \,||\, E_2.code \,||$

$x \,||$ ’:=’ $||\, E_2.rval \,||$ ’*’ $||\, E_1.elemsize \,||$

$E.lval \,||$ ’:=’ $||\, E_1.lval \,||$ ’+’ $||\, x$ }

c[i]

$E \longrightarrow E_1$. id { // for field access

$E.lval = newtemp();$

$E.lcode = E_1.lcode \,||$

$E.lval \,||$ ’:=’ $||\, E_1.lval \,||$ ’+’ $||$ id.$offset$ }

struct x {
int a;
float b;
} x;

x.a resides at the same location as x.

x.b resides at location of x + sizeof(int)

Computing lval and rval attributes

$E \longrightarrow E_1 = E_2$ {

$\quad E.code = E_1.lcode \,||\, E_2.code \,||$

$\quad\quad gen(\text{'*'}\ E_1.lval\ \text{':='}\ E_2.rval)$

$\quad E.rval = E_2.rval$ }

$E \longrightarrow E_1\ [\ E_2\ ]$ {

$\quad E.lval = newtemp();$

$\quad E.rval = newtemp();$

$\quad x = newtemp();$

$\quad E.lcode = E_1.lcode \,||\, E_2.code \,||$

$\quad\quad gen(x\ \text{':='}\ E_2.rval * E_1.elemsize) \,||$

$\quad\quad gen(E.lval\ \text{':='}\ E_1.lval\ \text{'+'}\ x)$

$\quad E.code = E.lcode \,||$

$\quad\quad gen(E.rval\ \text{':='}\ \text{'*'}\ E.lval)$

}

$\boxed{*x} = y$

$\boxed{\text{store } y, x}$

$c[d[i]] = 5 + 3$

Function Calls (Call-by-Value)

$(a \to f)(3, 5)$

$$
\begin{aligned}
E \longrightarrow E_1 (E_2, E_3) \quad \{ \quad & E.rval = newtemp(); \\
& E.code = E_1.code \mid\mid \\
& \qquad E_2.code \mid\mid \\
& \qquad E_3.code \mid\mid \\
& \qquad gen(\texttt{push } E_2.rval) \\
& \qquad gen(\texttt{push } E_3.rval) \\
& \qquad gen(\texttt{call } E_1.rval) \\
& \qquad gen(\texttt{pop } E.rval) \\
\}
\end{aligned}
$$

also need to pop arguments

Function Calls (Call-by-Reference)

$$
\begin{aligned}
E \quad \longrightarrow E_1\ (\ \underline{E_2},\ \underline{E_3}\ )\ \{ \\
& E.rval = newtemp(); \\
& E.code = E_1.code \;||\; \\
& \qquad E_2.lcode \;||\; \\
& \qquad E_3.lcode \;||\; \\
& \qquad gen(\texttt{push}\ E_2.lval) \\
& \qquad gen(\texttt{push}\ E_3.lval) \\
& \qquad gen(\texttt{call}\ E_1.rval) \\
& \qquad gen(\texttt{pop}\ E.rval) \\
\}
\end{aligned}
$$

Code Generation for Statements

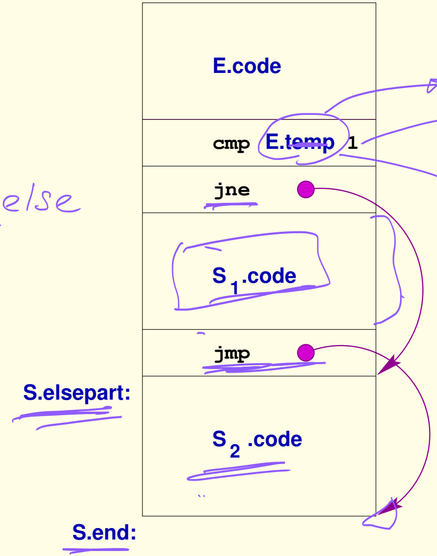
$$
\begin{array}{llll}
S & \longrightarrow & S_1 \,;\, S_2 & \{ \\
 & & & \quad S.code = S_1.code \;\|\; \\
 & & & \qquad\qquad S_2.code; \\
 & & & \} \\
 & & & \\
S & \longrightarrow & E & \{\, S.code = E.code; \,\}
\end{array}
$$

Conditional Statements

then

else

$$S \longrightarrow \text{if } E, S_1, S_2$$



represents "true"

previously

called E.addr

Conditional Statements

$$
\begin{aligned}
S \quad \longrightarrow \quad & \textsf{if } E, S_1, S_2 \quad \{ \\
& \quad \mathit{elselabel} = \mathit{newlabel}(); \\
& \quad \mathit{endlabel} = \mathit{newlabel}(); \\
& \quad S.\mathit{code} = \qquad E.\mathit{code} \; || \\
& \qquad\qquad\qquad \mathit{gen}(\textsf{if } E.\mathit{temp} \text{ `}\neq\text{' `1' } \mathit{elselabel}) \; || \\
& \qquad\qquad\qquad S_1.\mathit{code} \; || \\
& \qquad\qquad\qquad \mathit{gen}(\textsf{jmp } \mathit{endlabel}) \; || \\
& \qquad\qquad \mathit{gen}(\mathit{elselabel}{:}) \; || \\
& \qquad\qquad\qquad S_2.\mathit{code} \; || \\
& \qquad\qquad \mathit{gen}(\mathit{endlabel}{:}) \\
& \}
\end{aligned}
$$

L1

L2

L3

"if" ||

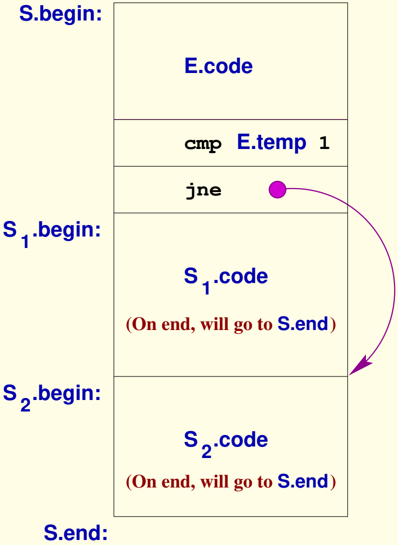
E.temp ||

"≠" ||

elselabel

If Statements: An Alternative

$$S \longrightarrow \text{if } E, S_1, S_2$$



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***.

Conditional Statements and Continuations

$$
\begin{aligned}
S \quad \longrightarrow \quad & \text{if } E, S_1, S_2 \quad \{ \\
& \quad S.begin = newlabel(); \\
& \quad S.end = newlabel(); \\
& \quad S_1.end = S_2.end = S.end; \\
& \quad S.code = gen(S.begin:) \; || \\
& \qquad\qquad E.code \; || \\
& \qquad\qquad gen(\text{if } E.rval \text{ '==' '1' } S_2.begin) \; || \\
& \qquad\qquad S_1.code \; || \\
& \qquad\qquad S_2.code; || \\
& \qquad\qquad gen(S.end:) \\
& \}
\end{aligned}
$$

Continuations

- Each boolean expression has two possible continuations:
  - $E.true$: where control will go when expression in E evaluates to *true*.
  - $E.false$: where control will go when expression in E evaluates to *false*.
- Every statement S has one continuation, $S.next$
- Every while loop statement has an additional continuation, $S.begin$

Shortcircuit Code for Boolean Expressions

$$
\begin{aligned}
\boxed{E} \quad \longrightarrow & \; E_1 \;\&\&\; E_2 \; \{ \\
& E_1.true = newlabel(); \\
& E_1.false = E_2.false = E.false; \\
& E_2.true = E.true; \\
& E.code = E_1.code \,||\, gen(E_1.true `\!:\!') \,||\, E_2.code \\
\}
\end{aligned}
$$

$E \rightarrow$ comparison

$$
\begin{aligned}
E \quad \longrightarrow & \; E_1 \text{ or } E_2 \; \{ \\
& E_1.true = E_2.true = E.true; \\
& E_1.false = newlabel(); \\
& E_2.false = E.false; \\
& E.code = E_1.code \,||\, gen(E_1.false `\!:\!') \,||\, E_2.code \\
\}
\end{aligned}
$$

$$
\begin{aligned}
E \quad \longrightarrow & \; ! \; E_1 \; \{ \\
& E_1.false = E.true; \; E_1.true = E.false; \\
\}
\end{aligned}
$$

$$
E \quad \longrightarrow \; \text{true} \; \{ \; E.code = gen(\text{jmp}, \; E.true) \; \}
$$

Short-circuit code for Conditional Statements

$S \longrightarrow S_1 ; S_2$ {

$S_1.next = newlabel();$

$S.code = S_1.code \,||\, gen(S_1.next \text{ ':'}) \,||\, S_2.code;$

}

$S \longrightarrow$ if E then S_1 else S_2 {

$E.true = newlabel();$

$E.false = newlabel();$

$S_1.next = S_2.next = S.next;$

$S.code = E.code \,||$

$gen(E.true \text{ ':'}) \,||\, S_1.code \,||$

$gen(\text{jmp } S.next) \,||$

$gen(E.false \text{ ':'}) \,||\, S_2.code;$

}

Continuation-style code for if-then-else

Short-circuit code for While

$$
\begin{aligned}
S &\longrightarrow \text{while } E \text{ do } S_1 \; \{ \\
&S.begin = newlabel(); \\
&E.true = newlabel(); \\
&E.false = S.next; \\
&S_1.next = S.begin; \\
&S.code = gen(S.begin\text{`:'}) \; || \; E.code \; || \\
&\qquad gen(E.true\text{`:'}) \; || \; S_1.code \; || \\
&\qquad gen(\text{jmp } S.begin); \\
&\}
\end{aligned}
$$

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.

Machine Code Generation Issues

- Register assignment
- Instruction selection
- ...

How GCC Handles Machine Code Generation

- gcc uses machine descriptions to *automatically* generate code for target machine
- machine descriptions specify:
  - memory addressing (bit, byte, word, big-endian, ...)
  - registers (how many, whether general purpose or not, ...)
  - stack layout
  - parameter passing conventions
  - semantics of instructions
  - . . .

Specifying Instruction Semantics

- gcc uses intermediate code called RTL, which uses a LISP-like syntax
- after parsing, programs are translated into RTL
- semantics of each instruction is also specified using RTL:

  `movl (r3), @8(r4)` $\equiv$

  ```
  (set (mem: SI (plus: SI (reg: SI 4) (const_int 8)))
       (mem: SI (reg: SI 3)))
  ```
- cost of machine instructions also specified
- gcc code generation = selecting a low-cost instruction sequence that has the same semantics as the intermediate code