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:

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

$E.code = E_1.code \parallel$

$E_2.code \parallel$

"add")

}

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"
- Three address code:

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

  Quadruples and Triples: Representations for three-address code.

Quadruples

Explicit representation of three-address code.
Example: `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.

Generating 3-address code

$E \longrightarrow E_1 + E_2$ {
  $E.temp = newtemp()$;
  $E.code = E_1.code \parallel E_2.code \parallel$
    $E.temp \parallel$ ':=' $\parallel E_1.temp \parallel$ '+' $\parallel E_2.temp$;
}
$E \longrightarrow$ int {
  $E.temp = newtemp()$;
  $E.code = E.temp \parallel$ ':=' $\parallel$ int.val;
}
$E \longrightarrow$ id {
  $E.temp =$ id.$name$;
  $E.code =$ '';
}

Generation of Postfix Code for Boolean Expressions

$E \longrightarrow E_1$ && E_2 {
  $E.code = E_1.code \parallel$
    $E_2.code \parallel$
    $gen($`and`$)$
}
$E \longrightarrow\ !\ E_1$ {
  $E.code = E_1.code \parallel$
    $gen($`not`$)$
}
$E \longrightarrow$ true $E.code = gen($`load_immed`, 1$)$
$E \longrightarrow$ id $E.code = gen($`load`, id.$addr)$

Code for Boolean Expressions

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

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

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

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

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

Computing l-values

$E \longrightarrow$ id {
  $E.lval$ = id.$name$;
  $E.code$ = '';
}
$E \longrightarrow E_1$ [E_2] {
  $E.lval$ = $newtemp()$;
  $E.lcode = E_1.lcode \parallel E_2.code \parallel$
    $E.lval \parallel$ ':=' $\parallel E_1.lval \parallel$ '+' $\parallel E_2.rval$
}
$E \longrightarrow E_1$. id { // for field access
  $E.lval$ = $newtemp()$;
  $E.lcode = E_1.lcode \parallel$
    $E.lval \parallel$ ':=' $\parallel E_1.lval \parallel$ '+' $\parallel$ id.$offset$
}

Computing lval and rval attributes

$E \longrightarrow E_1 = E_2$ {
$E.code = E_1.lcode \parallel E_2.code \parallel$
$gen(\text{'*'}\ E_1.lval\ \text{':='}\ E_2.rval)$
$E.rval = E_2.rval$
}
$E \longrightarrow E_1\ [\ E_2\ ]$ {
$E.lval = newtemp();$
$E.rval = newtemp();$
$E.lcode = E_1.lcode \parallel E_2.code \parallel$
$gen(E.lval\ \text{':='}\ E_1.lval\ \text{'+'}\ E_2.rval)$
$E.code = E.lcode \parallel$
$gen(E.rval\ \text{':= *'}\ E.lval)$
}

Function Calls (Call-by-Value)

$E \longrightarrow E_1\ (\ E_2,\ E_3\ )$ {
$E.rval = newtemp();$
$E.code = E_1.code \parallel$
$E_2.code \parallel$
$E_3.code \parallel$
$gen(\texttt{push}\ E_2.rval)$
$gen(\texttt{push}\ E_3.rval)$
$gen(\texttt{call}\ E_1.rval)$
$gen(\texttt{pop}\ E.rval)$
}

Function Calls (Call-by-Reference)

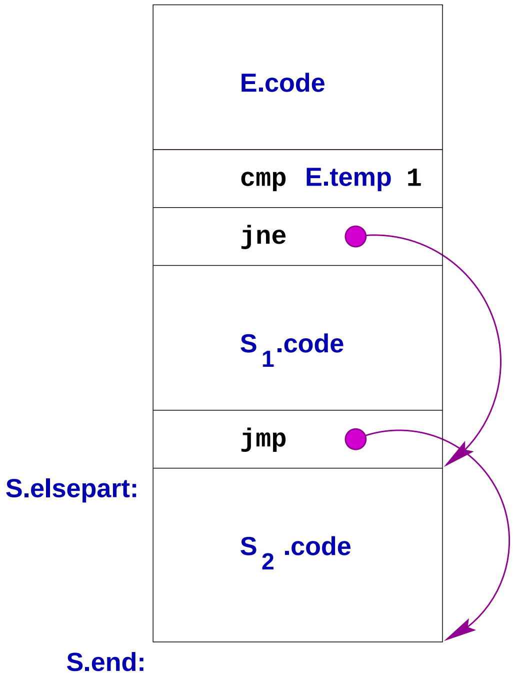
$E \longrightarrow E_1\ (\ E_2,\ E_3\ )$ {
$E.rval = newtemp();$
$E.code = E_1.code \parallel$
$E_2.lcode \parallel$
$E_3.lcode \parallel$
$gen(\texttt{push}\ E_2.lval)$
$gen(\texttt{push}\ E_3.lval)$
$gen(\texttt{call}\ E_1.rval)$
$gen(\texttt{pop}\ E.rval)$
}

Code Generation for Statements

$S \longrightarrow S_1\ ;\ S_2$ {
$S.code = S_1.code \parallel$
$S_2.code;$
}
$S \longrightarrow E$ { $S.code = E.code;$ }

Conditional Statements

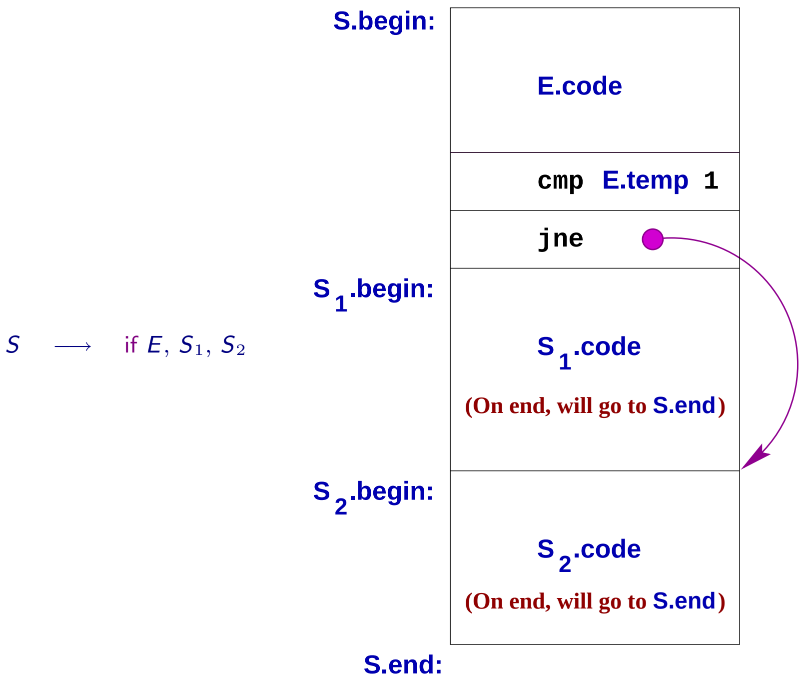
$S \longrightarrow \text{if } E,\, S_1,\, S_2$



Conditional Statements

$$
\begin{array}{lll}
S \longrightarrow \text{if } E,\, S_1,\, S_2 & \{ & \\
 & \mathit{elselabel} = \mathit{newlabel}(); & \\
 & \mathit{endlabel} = \mathit{newlabel}(); & \\
 & S.\mathit{code} = & E.\mathit{code} \parallel \\
 & & \mathit{gen}(\texttt{cmp } E.\mathit{temp},\, 1) \parallel \\
 & & \mathit{gen}(\texttt{jne } \mathit{elselabel}) \parallel \\
 & & S_1.\mathit{code} \parallel \\
 & & \mathit{gen}(\texttt{jmp } \mathit{endlabel}) \parallel \\
 & & \mathit{gen}(\mathit{elselabel}:) \parallel \\
 & & S_2.\mathit{code} \parallel \\
 & & \mathit{gen}(\mathit{endlabel}:) \\
 & \} &
\end{array}
$$

If Statements: An Alternative



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

$$
\begin{array}{lll}
S & \longrightarrow & \text{if } E,\ S_1,\ S_2 \quad \{ \\
 & & S.begin = newlabel(); \\
 & & S.end = newlabel(); \\
 & & S_1.end = S_2.end = S.end; \\
 & & S.code = gen(S.begin:) \parallel \\
 & & \qquad E.code \parallel \\
 & & \qquad gen(\texttt{cmp}\ E.place,\ 1) \parallel \\
 & & \qquad gen(\texttt{jz}\ S_2.begin) \parallel \\
 & & \qquad S_1.code \parallel \\
 & & \qquad S_2.code; \parallel \\
 & & \qquad gen(S.end:) \\
 & & \}
\end{array}
$$

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

$E \longrightarrow E_1$ && E_2 {
$E_1.true = newlabel();$
$E_1.false = E_2.false = E.false;$
$E_2.true = E.true;$
$E.code = E_1.code \parallel gen(E_1.true`:') \parallel E_2.code$
}
$E \longrightarrow E_1$ or E_2 {
$E_1.true = E_2.true = E.true;$
$E_1.false = newlabel();$
$E_2.false = E.false;$
$E.code = E_1.code \parallel gen(E_1.false`:') \parallel E_2.code$
}
$E \longrightarrow\ !\ E_1$ {
$E_1.false = E.true;\ E_1.true = E.false;$
}
$E \longrightarrow$ true { $E.code = gen(\texttt{goto}, E.true)$ }

Short-circuit code for Conditional Statements

$S \longrightarrow S_1\ ;\ S_2$ {
$S_1.next = newlabel();$
$S.code = S_1.code \parallel gen(S_1.next\ `:') \parallel S_2.code;$
}
$S \longrightarrow$ if E then S_1 else S_2 {
$E.true = newlabel();$
$E.false = newlabel();$
$S_1.next = S_2.next = S.next;$
$S.code = E.code \parallel$
$gen(E.true`:') \parallel S_1.code \parallel$
$gen(\text{`goto'}\ S.next) \parallel$
$gen(E.false`:') \parallel S_2.code;$
}

Short-circuit code for While

$S \longrightarrow$ while E do S_1 {
$S.begin = newlabel();$
$E.true = newlabel();$
$E.false = S.next;$
$S_1.next = S.begin;$
$S.code = gen(S.begin`:') \parallel E.code \parallel$
$gen(E.true`:') \parallel S_1.code \parallel$
$gen(\text{`goto'}\ S.begin);$
}

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 taget 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) ≡
    (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