**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:**

\[ E \rightarrow E_1 + E_2 \quad \{ \text{E.val := E}_1\text{.val} + E_2\text{.val;} \} \]

**Type Checking:**

\[ E \rightarrow E_1 + E_2 \quad \{
\text{if } E_1\text{.type }\equiv E_2\text{.type }\equiv \text{int}
\quad \text{E.type }= \text{int};
\text{else}
\quad \text{E.type }= \text{float};
\} \]

Coercion
Code Generation via Syntax Directed Translation

Print \[ 3 + (2 \times 5) \]

**Code Generation:**

\[ E \rightarrow E_1 + E_2 \]

\[
\begin{align*}
E\text{.code} &= E_1\text{.code} || \\
& \quad E_2\text{.code} || \\
& \quad \text{“add”}
\end{align*}
\]

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 $r_1, r_2$”

- **Three address code:**
  Code of the form “add $src_1, src_2, dest$”

**Quadruples and Triples:** Representations for three-address code.
Explicit representation of three-address code.

Example: \( a := a + b \times -c; \)

<table>
<thead>
<tr>
<th>Instr</th>
<th>Operation</th>
<th>Arg 1</th>
<th>Arg 2</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0)</td>
<td>uminus</td>
<td>c</td>
<td></td>
<td>( t_1 )</td>
</tr>
<tr>
<td>(1)</td>
<td>mult</td>
<td>b</td>
<td>( t_1 )</td>
<td>( t_2 )</td>
</tr>
<tr>
<td>(2)</td>
<td>add</td>
<td>a</td>
<td>( t_2 )</td>
<td>( t_3 )</td>
</tr>
<tr>
<td>(3)</td>
<td>move</td>
<td>( t_3 )</td>
<td></td>
<td>a</td>
</tr>
</tbody>
</table>
Triples

Representation of three-address code with implicit destination argument.
Example: \( a := a + b * -c; \)

<table>
<thead>
<tr>
<th>Instr</th>
<th>Operation</th>
<th>Arg 1</th>
<th>Arg 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0)</td>
<td>uminus</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>(1)</td>
<td>mult</td>
<td>b</td>
<td>(0)</td>
</tr>
<tr>
<td>(2)</td>
<td>add</td>
<td>a</td>
<td>(1)</td>
</tr>
<tr>
<td>(3)</td>
<td>move</td>
<td>a</td>
<td>(2)</td>
</tr>
</tbody>
</table>
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
- $\phi$ nodes used to combine values of variables after a conditional

Original Code:

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

Becomes:

```c
if (f) x_1 = 1; else x_2=2;
x_3 = \phi(x_1, x_2);
y=x_3*x_3;
```
Generating 3-address code

\[
E \rightarrow E_1 + E_2 \\
E.\text{addr} = \text{newtemp}();
\]

\[
E.\text{code} = [E_1.\text{code}] || [E_2.\text{code}]
\]

\[
E.\text{addr} \rightarrow \mathbin{\text{'='}} || E_1.\text{addr} \rightarrow \mathbin{\text{'+'}} || E_2.\text{addr};
\]

\[
E \rightarrow \text{int}
\]

\[
E.\text{addr} = \text{newtemp}();
\]

\[
E.\text{code} = E.\text{addr} \rightarrow \mathbin{\text{':='}} || \text{int.val};
\]

\[
E \rightarrow \text{id}
\]

\[
E.\text{addr} = \text{id.name};
\]

\[
E.\text{code} = "\;
\]

// no code needed

### String concatenation 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 var.
Generation of Postfix Code for Boolean Expressions

\[ E \rightarrow E_1 \&\& E_2 \{
\text{\texttt{E.code}} = \text{\texttt{E_1.code}} |\|\text{\texttt{E_2.code}} |\| \text{\texttt{gen(and)}} \}
\]

\[ E \rightarrow \texttt{! E_1} \{
\text{\texttt{E.code}} = \text{\texttt{E_1.code}} |\| \text{\texttt{gen(not)}} \}
\]

\[ E \rightarrow \texttt{true} \{ \text{\texttt{E.code}} = \text{\texttt{gen(load\_immed, 1)}} \}
\]

\[ E \rightarrow \texttt{id} \{ \text{\texttt{E.code}} = \text{\texttt{gen(load, id.addr)}} \} \]
Code for Boolean Expressions

```c
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

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

```c
load(p);
null();
neq();
jnz elselabel;
load(p);
ildc(1);
getfield();
load(q);
neq();
jnz elselabel;
... then part
easelabel:
... else part
```
**l- and r-Values**

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

- **r-value**: actual value of the expression

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

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

```
a[i]
```
Computing $l$-values

$E \to id \{ \\
E.lval = id.loc; \\
E.code = "\"; \}
$

// empty code

$E \to [E_1] E_2 \} \\
E.lval = newtemp(); \\
x = newtemp(); \\
E.lcode = E_1.lcode \mid E_2.code \mid \\
E.lval \mid ':=\' \mid E_2.rval \mid '+' \mid E_1.elemsize \mid \\
x \mid ':=\' \mid E_1.lval \mid '+' \mid x \}$

$E \to E_1.id \{ // for field access \\
E.lval = newtemp(); \\
E.lcode = E_1.lcode \mid \\
E.lval \mid ':=\' \mid E_1.lval \mid '+' \mid id.offset \}$

The loc attribute stores the location of the variable id.
Computing lval and rval attributes

\[ E \rightarrow E_1 = E_2 \{
\]
\[ E\.\text{code} = E_1\.\text{lcode} || E_2\.\text{code} || \]
\[ \text{gen}('**' E_1\.lval ':=' E_2\.rval) \]
\[ E\.\text{rval} = E_2\.\text{rval} \} \]

\[ E \rightarrow E_1 [ E_2 ] \{
\]
\[ E\.\text{lval} = \text{newtemp}(); \]
\[ E\.\text{rval} = \text{newtemp}(); \]
\[ x = \text{newtemp}(); \]
\[ E\.\text{lcode} = E_1\.\text{lcode} || E_2\.\text{code} || \]
\[ \text{gen}(x '::=' E_2\.rval*E_1\.\text{elemsize}) || \]
\[ \text{gen}(E\.\text{lval} '::=' E_1\.\text{lval} '+' x) \]
\[ E\.\text{code} = E\.\text{lcode} || \]
\[ \text{gen}(E\.\text{rval} '::=' '***' E\.\text{lval}) \}

\[ e[a[i]] = 5 + 3 \]
Function Calls (Call-by-Value)

\[ E \rightarrow E_1(E_2, E_3) \]

\[
E.\text{rval} = \text{newtemp}();
\]

\[
E.\text{code} = E_1.\text{code} \ |
\]

\[
E_2.\text{code} \ |
\]

\[
E_3.\text{code} \ |
\]

\[
\text{gen(push } E_2.\text{rval})
\]

\[
\text{gen(push } E_3.\text{rval})
\]

\[
\text{gen(call } E_1.\text{rval})
\]

\[
\text{gen(pop } E.\text{rval})
\]

\[
(a - 0.5)(3, 5)
\]

Also need to pop arguments.
Function Calls (Call-by-Reference)

\[ E \rightarrow E_1( E_2, E_3 ) \{ \]

\[ E.rval = \text{newtemp}(); \]
\[ E.code = E_1.code \|
\]
\[ E_2.lcode \|
\]
\[ E_3.lcode \|
\]
\[ \text{gen}(\text{push } E_2.lval) \]
\[ \text{gen}(\text{push } E_3.lval) \]
\[ \text{gen}(\text{call } E_1.rval) \]
\[ \text{gen}(\text{pop } E.rval) \]
\}
Code Generation for Statements

\[ S \rightarrow S_1 ; S_2 \{ \]
\[ \quad S.code = S_1.code || S_2.code; \]
\[ \}

\[ S \rightarrow E \{ S.code = E.code; \} \]
Conditional Statements

\[ S \rightarrow \text{if } E, S_1, S_2 \]

- \text{if} \ E
- \text{else:} \ S_2 \ . \ code
- \text{then:} \ S_1 \ . \ code

- cmp \ E \ . \ temp 1
- jne
- jmp

\( \text{cmp } \) represents "true"
\( \text{jmp} \) previously called \( E \ . \ addr \)
Conditional Statements

\[ S \rightarrow \text{if } E, S_1, S_2 \{ \]

\[
\text{elselabel} = \text{newlabel}();
\]

\[
\text{endlabel} = \text{newlabel}();
\]

\[
S.\text{code} = E.\text{code} |\|
\]

\[
\text{gen}(\text{if } E.\text{temp} \neq '1' \text{ elselabel}) |\|
\]

\[
S_1.\text{code} |\|
\]

\[
\text{gen}(\text{jmp endlabel}) |\|
\]

\[
\text{gen}(\text{elselabel:}) |\|
\]

\[
S_2.\text{code} |\|
\]

\[
\text{gen}(\text{endlabel:})
\}

\[ L_1: \]

\[ L_2: \text{"if" } 11 \]

\[ L_3: \text{"\neq" } 11 \]

\[ \text{elselabel} \]
If Statements: An Alternative

\[ S \quad \rightarrow \quad \text{if } E, S_1, S_2 \]
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

\[ S \rightarrow \text{if}(E, S_1, S_2) \{ \]

\[ S\.begin = \text{newlabel}(); \]
\[ S\.end = \text{newlabel}(); \]
\[ S_1\.end = S_2\.end = S\.end; \]
\[ S\.code = \text{gen}(S\.begin:) || \]
\[ E\.code || \]
\[ \text{gen}(\text{if } E\.rval \text{ '==' '1'} S_2\.begin) || \]
\[ S_1\.code || \]
\[ S_2\.code; || \]
\[ \text{gen}(S\.end:) \]
\}
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

\[
E \rightarrow E_1 \&\& E_2 \{
E_1.true = \text{newlabel}();
E_1.false = E_2.false = E.false;
E_2.true = E.true;
E.code = E_1.code || \text{gen}(E_1.true'') \| E_2.code
\}
\]

\[
E \rightarrow E_1 \text{ or } E_2 \{
E_1.true = E_2.true = E.true;
E_1.false = \text{newlabel}();
E_2.false = E.false;
E.code = E_1.code || \text{gen}(E_1.false'') \| E_2.code
\}
\]

\[
E \rightarrow ! E_1 \{
E_1.false = E.true; E_1.true = E.false;
\}
\]

\[
E \rightarrow \text{true} \{ E.code = \text{gen}(\text{jmp}, E.true) \}
\]
Short-circuit code for Conditional Statements

\[ S \rightarrow S_1; S_2 \{
S_1.next = \text{newlabel}();
S.code = S_1.code || \text{gen}(S_1.next ':') || S_2.code;
\}
\]

\[ S \rightarrow \text{if } E \text{ then } S_1 \text{ else } S_2 \{
E.true = \text{newlabel}();
E.false = \text{newlabel}();
S_1.next = S_2.next = S.next;
S.code = E.code || \text{gen}(E.true ':') || S_1.code ||
\text{gen}(\text{jmp } S.next) ||
\text{gen}(E.false ':') || S_2.code;
\}

Continuation-style code for if-then-else
$S \rightarrow$ while $E$ do $S_1$

\[
S.\text{begin} = \text{newlabel}(); \\
E.\text{true} = \text{newlabel}(); \\
E.\text{false} = S.\text{next}; \\
S_1.\text{next} = S.\text{begin}; \\
S.\text{code} = \text{gen}(S.\text{begin}'') \mid| E.\text{code} \mid| \\
\text{gen}(E.\text{true}'') \mid| S_1.\text{code} \mid| \\
\text{gen}(\text{jmp} S.\text{begin});
\]

\[\]

\[
\]
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
- ...

30 / 32
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:
  \[
  \text{movl} \ (r3), \ @8(r4) \equiv \\
  (\text{set} \ (\text{mem}: \ SI \ (\text{plus}: \ SI \ (\text{reg}: \ SI \ 4) \ (\text{const} \ _\text{int} \ 8)))) \\
  (\text{mem}: \ SI \ (\text{reg}: \ SI \ 3))
  \]
- cost of machine instructions also specified
- gcc code generation = selecting a low-cost instruction sequence that has the same semantics as the intermediate code