

## 8. Intermediate Code

- Intermediate code is closer to the target machine than the source language, and hence easier to generate code from.
- Unlike machine language, intermediate code is (more or less) machine independent. This makes it easier to retarget the compiler.
- It allows a variety of optimizations to be performed in a machine-independent way.
- Typically, intermediate code generation can be implemented via syntax-directed translation, and thus can be folded into parsing by augmenting the code for the parser.

## 8.1.1. Low-Level Intermediate Representations



### Examples : Three Address Code

- This is a sequence of instructions of the form

$$x := y \text{ op } z$$

where  $x$ ,  $y$ , and  $z$  are *variable names*, *constants*, or *compiler generated variables* (“temporaries”).

- Only one operator is permitted on the RHS, so there are no “built-up” expressions. Instead, expressions are computed using temporaries. E.g. the source language construct

$$x := y + z * w$$

might translate to

$$\begin{aligned} t1 &:= z * w \\ x &:= y + t1 \end{aligned}$$

## Different Kinds of Three-Address Statements

### Assignment :

$x := y \text{ op } z,$                       *op* binary  
 $x := \text{op } y,$                               *op* unary  
 $x := y$

### Jumps :

$\text{goto } L,$   
 $\text{jumpr } t \text{ } L,$   
 $\text{jumpr } t \text{ } L,$                               *L* a label  
  
 $\text{if } x \text{ relop } y \text{ goto } L, \text{ } L \text{ a label}$

## Procedure Call/Return :

param $x$ ,	$x$ an actual parameter
call $p, n$ ,	$n =$ no. of params to $p$
enter	initialization (if any)
exit	cleanup actions (if any)
return	
return $x$	
retrieve $x$	save returned value in $x$

## Indexed Assignment :

$x := y[i]$   
 $x[i] := y$

## Address and Pointer Assignments :

$x := \&y$   
 $x := *y$   
 $x := y$

## Miscellaneous :

label  $L$

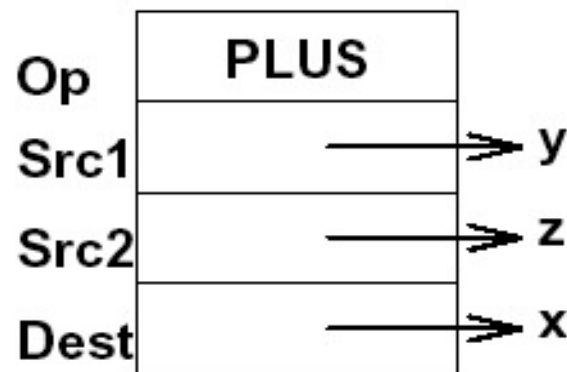
## 8.1.2. Implementing Three-Address Instructions

Each instruction is implemented as a structure called a *quadruple*:

- contains (upto) 4 fields: operation, (upto) two operands, and destination;
- for operands: use a bit to indicate whether it's a constant or a pointer into the symbol table.

`x := y + z`

`if t1 >= t2 goto L`



instruction  
labelled L

## 8.2. Intermediate Code Generation

- Source language constructs are decomposed to simpler constructs at the intermediate code level.
- When generating code to evaluate expressions, temporary names must be made up for internal nodes in the syntax tree for the expression.

Example:

**Source** : `if  $x + 2 > 3 * (y - 1) + 4$  then  $z := 0$ ;`

**Intermediate Code** :

```

t1 := x+2
t2 := y-1
t3 := 3*t2
t4 := t3+4
if t1 <= t4 goto L
z := 0
label L

```

## Intermediate Code Generation

### Syntax-Directed Translation :

- Intermediate code represented as a list of instructions. Instruction sequences are concatenated using the operator `||`.

(In practice, we might choose to write the intermediate code instructions out into a file.)

- Attributes for Expressions  $E$  :
  - $E.place$  : denotes the location that holds the value of  $E$ .
  - $E.code$  : denotes the instruction sequence that evaluates  $E$ .



- Attributes for Statements  $S$  :
  - $S.begin$  : denotes the first instruction in the code for  $S$ .
  - $S.after$  : denotes the first instruction *after* the code for  $S$ .
  - $S.code$  : denotes the instruction sequence that represents  $S$ .

- Auxiliary Functions :
  - newtemp() : returns a *new* temporary each time it is called.
    - \* returns a pointer to the ST entry of a temp.
    - \* may take a parameter specifying the type of the temp (useful if reusing temps).
  - newlabel() : returns a new label name each time it is called.
  
- Notation : we write
 
$$\text{gen}(x \text{ ':=' } y \text{ '+' } z)$$
 to represent the instruction  $x := y + z$ .

## Intermediate Code Generation : Simple Expressions



<u>PRODUCTION</u>	<u>SEMANTIC RULE</u>
$E \longrightarrow \text{id}$	$E.place := \text{id}.place;$ $E.code := '' ;$
$E \longrightarrow ( E_1 )$	$E.place := E_1.place;$ $E.code := E_1.code;$
$E \longrightarrow E_1 + E_2$	$E.place := \text{newtemp}();$ $E.code := E_1.code \parallel$ $E_2.code \parallel$ $\text{gen}(E.place ':='$ $E_1.place '+'$ $E_2.place)$
$E \longrightarrow -E_1$	$E.place := \text{newtemp}();$ $E.code := E_1.code \parallel$ $\text{gen}(E.place ':='$ $'-' E_1.place)$

### 8.3.2. Accessing Array Elements I

- Array elements can be accessed quickly if the elements are stored in a block of consecutive locations.
- Assume:
  - we want the  $i^{th}$  element of an array  $A$  whose subscript ranges from  $lo$  to  $hi$ ;
  - the address of the first element of the array is  $base$ .
- We can avoid address computations in the intermediate code if we have indexed “addressing modes” at the intermediate code level.

In this case,  $A[i]$  is the  $(i - lo)^{th}$  element of the array located at  $base$  (starting at element 0). So a reference  $A[i]$  translates to the code

```
t1 := i - lo
t2 := A[t1]
```

### 8.3.2. Accessing Array Elements II

- Address computations can't be avoided in general, because of pointer and `struct` types.
- The size of array element can be greater than 1 say  $w$ .
- Assume:
  - we want the  $i^{th}$  element of an array  $A$  whose subscript ranges from  $lo$  to  $hi$ ;
  - the address of the first element of the array is  $base$ ;
  - each element of  $A$  has width  $w$ .

Then, the address of  $x[i]$  is

$$\begin{aligned} & \text{base} + (i - lo) * w \\ &= (\text{base} - lo * w) + i * w \\ &= C_A + i * w \end{aligned}$$

where  $C_A$  depends on the array  $A$  and is known at compile time.

Note :  $C_A$  is a memory address if  $A$  is a global, and is a stack displacement if  $A$  is a local.

- The idea extends to multidimensional arrays in the obvious way: need to know whether the elements are stored in row-major or column-major order.

## 8.4. Logical Expressions

$BExp \longrightarrow E_1 \text{ relop } E_2$

### 8.4.3. Naive but Simple Approach :

Intermediate Code (TRUE == 1, FALSE == 0) :

```

[
  t1 ← value of  $E_1$ 
]
[
  t2 ← value of  $E_2$ 
]
t3 := TRUE
if t1 relop t2 goto L
t3 := FALSE
label L

```

Disadvantage : Lots of (usually unnecessary) memory traffic.

## 8.4.1. Code Generation for Conditionals

Production :  $S \longrightarrow \text{if } E \text{ then } S_1 \text{ else } S_2$

Semantic Rule :

```

{ S.begin := newlabel();
  S.after := newlabel();
  S.code := gen('label' S.begin) ||
    E.code ||
    gen('if' E.place = "0" goto' S_2.begin) ||
    S_1.code ||
    gen('goto' S.after) ||
    S_2.code ||
    gen('label' S.after)
}

```

Structure of Generated Code

<p>L1: Code for E              if E == FALSE goto L3              Code for S1              goto L2          L3: Code for S2          L2: ...</p>
--



### 8.4.1. Code Generation for Loops

Production :  $S \longrightarrow \text{while } E \text{ do } S_1$

Structure of Generated Code :

```

L1 :
    [ Code for E
    if (E == FALSE) goto L2
    [ Code for S1
    goto L1
L2 : ...
  
```

Semantic Rule :

```

{ S.begin := newlabel();
  S.after := newlabel();
  S.code := gen('label' S.begin) ||
            E.code ||
            gen('if' E.place = "0" goto' S.after) ||
            S1.code ||
            gen('goto' S.begin) ||
            gen('label' S.after)
}
  
```

## Intermediate Code Generation: Assignment

- Grammar productions:

$$S \longrightarrow Lhs \ := \ Rhs$$

- Semantic Rule:

$$\{ S.code \ := \ Lhs.code \ || \\ Rhs.code \ || \\ gen(Lhs.place \ ':= \ ' \ Rhs.place) \ }$$

## Relational Expressions: Better Approach

- Often, relational expressions occur in the context of boolean conditions of **control** statements.
- Instead of creating temporaries which are set to true or false, based upon the outcome of evaluating a boolean condition, generate direct branches to true and false targets.
- Short circuit evaluation of boolean expressions can also be handled effectively by this approach.

## Relational Expressions: Example

$E = a < b \text{ or } c < d \text{ and } e < f$

```

100 : if  $a < b$  goto --
101 : goto 102
102 : if  $c < d$  goto 104
103 : goto --
104 : if  $e < f$  goto --
105 : goto --

```

$E.truelist = \{100, 104\}$   
 $E.falselist = \{103, 105\}$

$$\begin{aligned} E &\longrightarrow E_1 \text{ or } M E_2 \\ &\{ \\ &\quad \text{backpatch}(E_1.\text{falselist}, M.\text{quad}); \\ &\quad E.\text{truelist} = \text{merge}(E_1.\text{truelist}, E_2.\text{truelist}); \\ &\quad E.\text{falselist} = E_2.\text{falselist}; \\ &\} \\ M &\longrightarrow \epsilon \\ &\{M.\text{quad} = \text{nextquad}\} \\ E &\longrightarrow E_1 \text{ and } M E_2 \\ &\{ \\ &\quad \text{backpatch}(E_1.\text{truelist}, M.\text{quad}); \\ &\quad E.\text{truelist} = E_2.\text{truelist}; \\ &\quad E.\text{falselist} = \text{merge}(E_1.\text{falselist}, E_2.\text{falselist}); \\ &\} \end{aligned}$$

## Relational Expressions: cont'd.



$E \longrightarrow \text{not } E_1$

```
{  
   $E.\text{truelist} = E_1.\text{falselist};$   
   $E.\text{falselist} = E_1.\text{truelist};$   
}
```

$E \longrightarrow (E_1)$

```
{  
   $E.\text{truelist} = E_1.\text{truelist};$   
   $E.\text{falselist} = E_1.\text{falselist};$   
}
```

$E \longrightarrow id_1 \text{ relop } id_2$

```
{  
   $E.\text{truelist} = \text{makelist}(\text{nextquad});$   
   $E.\text{falselist} = \text{makelist}(\text{nextquad} + 1);$   
   $\text{generate}(\text{if } id_1.\text{addr} \text{ relop } id_2.\text{addr} \text{ goto } \_)$   
   $\text{generate}(\text{goto } \_)$   
   $\_$   
}
```

```
E → true
  {
    E.truelist = makelist ( nextquad );
    generate ( goto__ )
  }
```

```
E → false
  {
    E.falselist = makelist ( nextquad );
    generate ( goto__ )
  }
```

## Code Generation for Loops and Conditionals



- Straightforward approach can introduce branch instructions whose targets are unconditional jumps.

```
while a < b do  
    if x < y then S endif  
endwhile
```

```
100: if a < b go to 102
```

```
101: go to 106
```

```
102: if x < y go to 104
```

```
103: go to 105 100
```

```
104: S.code
```

```
105: go to 100
```

```
106:
```

- We can avoid this by maintaining an additional attribute for statements called the *nextlist*. This attribute tracks branches in the statements whose target should be set to code that follows them in the execution sequence.



## Loops and Conditionals: cont'd.

```
S  $\longrightarrow$  if E then M1 S1 N else M2 S2
  {
  backpatch(E.truelist, M1.quad);
  backpatch(E.falselist, M2.quad);
  S.nextlist = merge(S1.nextlist, merge(N.nextlist, S2.nextlist))
  }
N  $\longrightarrow$   $\epsilon$ 
  {
  N.nextlist = makelist(nextquad);
  generate(goto___ )
  }
M  $\longrightarrow$   $\epsilon$ 
  {M.quad = nextquad}
S  $\longrightarrow$  if E then M S1
  {
  backpatch(E.truelist, M.quad);
  S.nextlist = merge(E.falselist, S1.nextlist)
  }
```

## Loops and Conditionals: cont'd.

$S \rightarrow \text{while } M_1 \ E \ \text{do } M_2 \ S_1$

```
{
backpatch(S1.nextlist, M1.quad);
backpatch(E.truelist, M2.quad);
S.nextlist = E.falselist;
generate(goto M1.quad);
}
```

**Optimization  
takes place here**



$S \rightarrow \text{begin } L \ \text{end}$

```
{S.nextlist = L.nextlist}
```

$S \rightarrow A$

```
{S.nextlist = nil}
```

$L \rightarrow L_1 ; M \ S$

```
{
backpatch(L1.nextlist, M.quad);
L.nextlist = S.nextlist;
}
```

$L \rightarrow S$

```
{L.nextlist = S.nextlist}
```

```
while a<b do  
    while x<y do  
        S  
    endwhile  
endwhile
```

```
100: if a < b go to 102  
101: go to 107  
102: if x < y go to 104  
103: go to 106 100  
104: S.code  
105: go to 102  
106: go to 100  
107: .....
```

## Intermediate Code Generation: case Statements

Implementation issue : Need to generate code so that we can (efficiently) choose one of a set of different alternatives, depending on the value of an expression.

### Implementation choices :

1. linear search
2. binary search
3. jump table

## Implementation considerations :

1. Execution Cost : linear or binary search may be cheaper if the no. of cases is small.  
For a large no. of cases, a jump table may be cheaper.
2. Space cost : a jump table may take too much space if the case values are not clustered closely together, e.g.:

```
switch (x) {  
    case 1 : ...  
    case 1000 : ...  
    case 1000000 : ...  
}
```

## 8.5. Code Generation for Function Calls

### Calling Sequence: Caller :

- Evaluate actual parameters; place actuals where the callee wants them.

*Instruction* : param  $t$

- Save machine state (current stack and/or frame pointers, return address) and transfer control to callee.

*Instruction* : call  $p, n$  ( $n =$  no. of actuals)

### Calling Sequence: Callee :

- Save registers (if necessary); update stack and frame pointers to accommodate  $m$  bytes of local storage.

*Instruction* : enter  $m$ .

### Return Sequence: Callee :

- Place return value  $x$  (if any) where the caller wants it; adjust stack/frame pointers (maybe); jump to return address.

*Instruction* : `return  $x$  Or return.`

### Return Sequence: Caller :

- Save the value returned by the callee (if any) into  $x$ .

*Instruction* : `retrieve  $x$ .`

## Intermediate Code for Function Calls: An Example

### Source Code :

```
x = f(0,y+1)-1;
```

### Intermediate Code Generated :

```
t1 := y+1
param t1      /* arg 2 */
param 0      /* arg 1 */
call f, 2
retrieve t2   /* t2 := f(0,t1) */
t3 := t2-1
x := t3
```

Suppose function *f* needs 24 bytes of space for its locals and temporaries. Its code has the form

```
enter 24
...
return t17
/* suppose return value is in t17 */
```



## Code Generation for Functions: Storage Allocation

**Problem** : The first instruction in a function is

```
enter n /* n = space for locals, temps */
```

but  $n$  is not known until the whole function has been processed.

**Solution 1** : generate final code into a list, “back-patch” the appropriate instructions after processing the function body.

Advantage : Can also do machine-dependent optimizations (e.g., instruction scheduling).

Disadvantage : slower, requires more memory.

Solution 2 : Generate code of the form

```

code
for
function
foo
    goto L1
    L2: [
        code for
        body of foo
    ]
    L1: [ code for enter n
        goto L2
    ]

```

## Reusing Temporaries

Storage requirements can be reduced considerably if we reuse temporaries:

- Maintain a free list of temporaries:
  - When a temporary is no longer necessary, it is returned to the free list.
  - The function *newtemp()* is modified to first search the free list, and to allocate a new temporary only if there is nothing in the free list.
  
- To handle objects of different sizes, we can maintain a free list for each type (or size).