Intermediate Code Generation - Part 2

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Outline of the Lecture

- Introduction (covered in part 1)
- Different types of intermediate code (covered in part 1)
- Intermediate code generation for various constructs

SATG for *If-Then-Else* Statement

- $IFEXP \rightarrow if \;\; E$
  { IFEXP.falselist := makelist(nextquad);
  gen('if E.result $\leq$ 0 goto __'); }
- $S \rightarrow IFEXP \;\; S_1 ; \;\; N \;\; else \;\; M \;\; S_2$
  { backpatch(IFEXP.falselist, M.quad);
  S.next := merge(S_1.next, S_2.next, N.next); }
- $S \rightarrow IFEXP \;\; S_1 ;$
  { S.next := merge(S_1.next, IFEXP.falselist); }
- $N \rightarrow \epsilon$
  { N.next := makelist(nextquad);
  gen('goto __'); }
- $M \rightarrow \epsilon$
  { M.quad := nextquad; }

SATG for Other Statements

- $S \rightarrow$ '{' L '}'
  { S.next := L.next; }
- $S \rightarrow A$
  { S.next := makelist(nil); }
- $S \rightarrow$ *return* E
  { gen('return E.result'); S.next := makelist(nil); }
- $L \rightarrow L_1$ ';' M S
  { backpatch(L_1.next, M.quad);
  L.next := S.next; }
- $L \rightarrow S$
  { L.next := S.next; }
- When the body of a procedure ends, we perform the following actions in addition to other actions:
  { backpatch(S.next, nextquad); gen('func end'); }

Translation Trace for *If-Then-Else* Statement

A_i are all assignments, and E_i are all expressions

if (E_1) { if (E_2) A_1; else A_2; }else A_3; A_4;

$S \Rightarrow IFEXP\ \ S_1;\ \ N_1\ \ else\ \ M_1\ \ S_2$

$\Rightarrow^* IFEXP_1\ \ IFEXP_2\ \ S_{21};\ \ N_2\ \ else\ \ M_2\ \ S_{22};\ \ N_1\ else\ \ M_1\ \ S_2$

1. Consider outer if-then-else
   Code generation for E_1
2. gen('if E_1.result $\leq$ 0 goto __')
   on reduction by $IFEXP_1 \rightarrow if\ \ E_1$
   Remember the above quad address in $IFEXP_1$.falselist
3. Consider inner if-then-else
   Code generation for E_2
4. gen('if E_2.result $\leq$ 0 goto __')
   on reduction by $IFEXP_2 \rightarrow if\ \ E_2$
   Remember the above quad address in $IFEXP_2$.falselist

Translation Trace for *If-Then-Else* Statement(contd.)

if (E_1) { if (E_2) A_1; else A_2; }else A_3; A_4;
$S \Rightarrow^* IFEXP_1\ IFEXP_2\ S_{21};\ N_2\ else\ M_2\ S_{22};\ N_1\ else\ M_1\ S_2$
Code generated so far:

Code for E_1; if E_1.result ≤ 0 goto __ (on $IFEXP_1$.falselist);
Code for E_2; if E_2.result ≤ 0 goto __ (on $IFEXP_2$.falselist);

5. Code generation for S_{21}
6. gen('goto __'), on reduction by $N_2 \rightarrow \epsilon$
   (remember in N_2.next)
7. L1: remember in M_2.quad, on reduction by $M_2 \rightarrow \epsilon$
8. Code generation for S_{22}
9. backpatch($IFEXP_2$.falselist, L1) (processing E_2 == false)
   on reduction by $S_1 \rightarrow IFEXP_2\ S_{21}\ N_2\ else\ M_2\ S_{22}$
   N_2.next is not yet patched; put on S_1.next

Translation Trace for *If-Then-Else* Statement(contd.)

if (E_1) { if (E_2) A_1; else A_2; }else A_3; A_4;
$S \Rightarrow IFEXP\ \ S_1;\ \ N_1\ \ else\ \ M_1\ \ S_2$
$S \Rightarrow^* IFEXP_1\ IFEXP_2\ S_{21};\ N_2\ else\ M_2\ S_{22};\ N_1\ else\ M_1\ S_2$
Code generated so far:
Code for E_1; if E_1.result ≤ 0 goto __ (on $IFEXP_1$.falselist)
Code for E_2; if E_2.result ≤ 0 goto L1
Code for S_{21}; goto __ (on S_1.next)
L1: Code for S_{22}

10. gen('goto __'), on reduction by $N_1 \rightarrow \epsilon$ (remember in N_1.next)
11. L2: remember in M_1.quad, on reduction by $M_1 \rightarrow \epsilon$
12. Code generation for S_2
13. backpatch(IFEXP.falselist, L2) (processing E_1 == false) on reduction by $S \rightarrow IFEXP\ \ S_1\ \ N_1\ \ else\ \ M_1\ \ S_2$
    N_1.next is merged with S_1.next, and put on S.next

Translation Trace for *If-Then-Else* Statement(contd.)

if (E_1) { if (E_2) A_1; else A_2; }else A_3; A_4;
$S \Rightarrow^*$ $IFEXP_1$ $IFEXP_2$ S_{21}; N_2 *else* M_2 S_{22}; N_1 *else* M_1 S_2
$L \Rightarrow^* L_1$ ';' M_3 S_4 $\Rightarrow^* S_3$ ';' M_3 S_4
Code generated so far (for S_3/L_1 above):

Code for E_1; if E_1.result $\leq$ 0 goto L2
Code for E_2; if E_2.result $\leq$ 0 goto L1
Code for S_{21}; goto __ (on S_3.next/L_1.next)
L1: Code for S_{22}
goto __ (on S_3.next/L_1.next)
L2: Code for S_2

14. L3: remember in M_3.quad, on reduction by $M_3 \rightarrow \epsilon$
15. Code generation for S_4
16. backpatch(L_1.next, L3), on reduction by $L \rightarrow L_1$ ';' M_3 S_4
17. L.next is empty

Translation Trace for *If-Then-Else* Statement(contd.)

if (E_1) { if (E_2) A_1; else A_2; }else A_3; A_4;
$S \Rightarrow^*$ $IFEXP_1$ $IFEXP_2$ S_{21}; N_2 *else* M_2 S_{22}; N_1 *else* M_1 S_2
$L \Rightarrow^* L_1$ ';' M_3 S_4 $\Rightarrow^* S_3$ ';' M_3 S_4

Final generated code

Code for E_1; if E_1.result ≤ 0 goto L2
Code for E_2; if E_2.result ≤ 0 goto L1
Code for S_{21}; goto L3
L1: Code for S_{22}
goto L3
L2: Code for S_2
L3: Code for S_4

SATG for *While-do* Statement

- $WHILEXEP \rightarrow while \;\; M \;\; E$
  { WHILEEXP.falselist := makelist(nextquad);
  gen('if E.result $\leq$ 0 goto __');
  WHILEEXP.begin := M.quad; }
- $S \rightarrow WHILEXEP \;\; do \;\; S_1$
  { gen('goto WHILEEXP.begin');
  backpatch(S_1.next, WHILEEXP.begin);
  S.next := WHILEEXP.falselist; }
- $M \rightarrow \epsilon$ (repeated here for convenience)
  { M.quad := nextquad; }

Code Template for *Function* Declaration and Call

Assumtion: No nesting of functions

```
result foo(parameter list){ variable declarations; Statement list; }
func begin foo
/* creates activation record for foo - */
/* - space for local variables and temporaries */
code for Statement list
func end /* releases activation record and return */

x = bar(p1,p2,p3);
code for evaluation of p1, p2, p3 (result in T1, T2, T3)
/* result is supposed to be returned in T4 */
param T1;  param T2;  param T3; refparam T4;
call bar, 4
/* creates appropriate access links, pushes return address */
/* and jumps to code for bar */
x = T4
```

SATG for *Function* Call

Assumtion: No nesting of functions

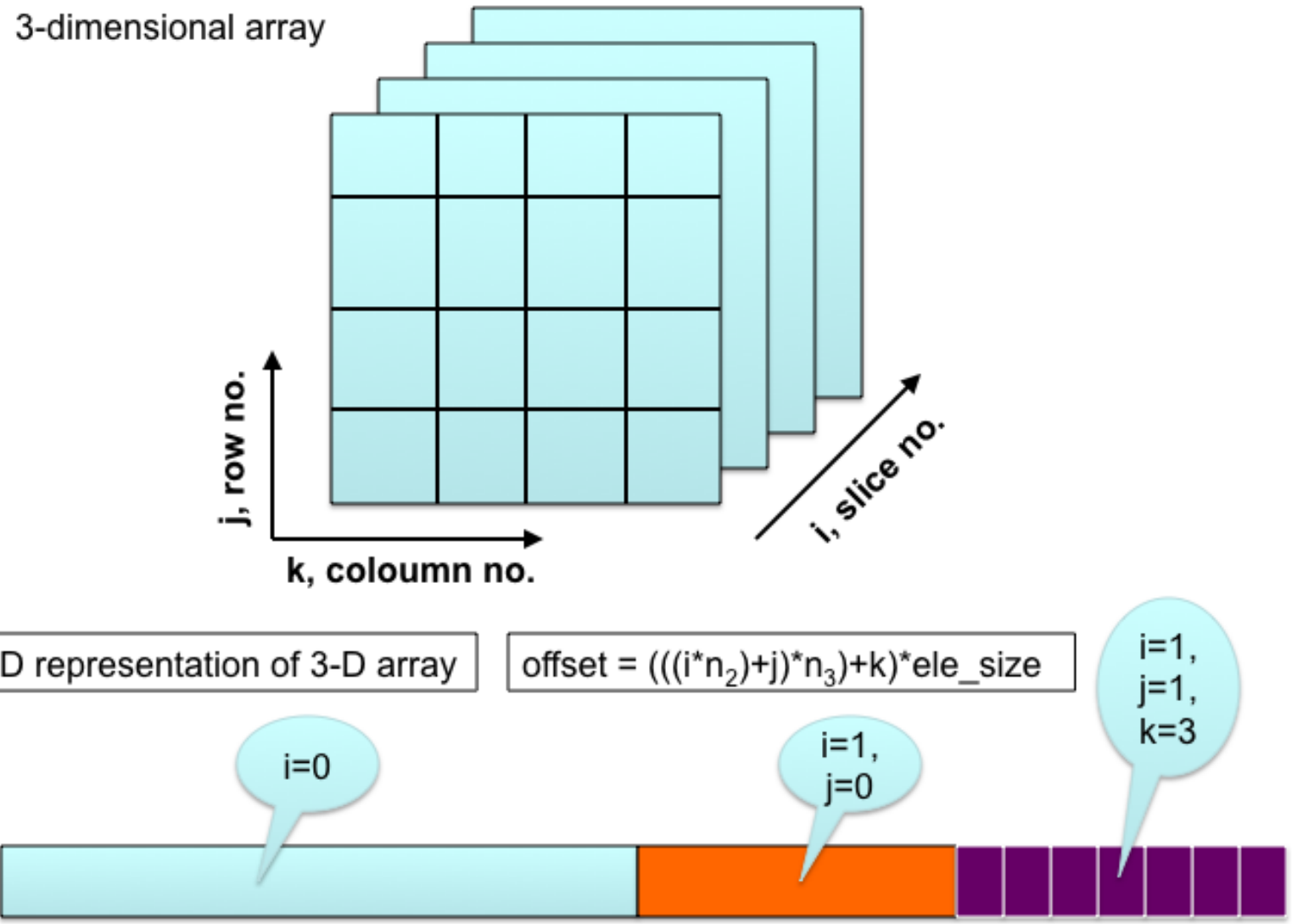
- *FUNC_CALL* $\rightarrow$ *id* **{action 1}** (*PARAMLIST*) **{action 2}**
  {action 1:} {search_func(id.name, found, fnptr);
  call_name_ptr := fnptr }
  {action 2:}
  { result_var := newtemp(get_result_type(call_name_ptr));
  gen('refparam result_var');
  /* Machine code for return a places a in result_var */
  gen('call call_name_ptr, PARAMLIST.pno+1'); }
- *PARAMLIST* $\rightarrow$ *PLIST* { PARAMLIST.pno := PLIST.pno }
- *PARAMLIST* $\rightarrow \epsilon$ {PARAMLIST.pno := 0 }
- *PLIST* $\rightarrow$ *E* { PLIST.pno := 1; gen('param E.result'); }
- $PLIST_1 \rightarrow PLIST_2$, *E*
  { $PLIST_1$.pno := $PLIST_2$.pno + 1; gen('param E.result'); }

SATG for *Function* Declaration

Assumtion: No nesting of functions

- *FUNC_DECL* $\rightarrow$ *FUNC_HEAD* { *VAR_DECL BODY* }
  { backpatch(BODY.next, nextquad);
  gen('func end');}
- *FUNC_HEAD* $\rightarrow$ *RESULT id* (*DECL_PLIST*)
  { search_func(id.name, found, namptr);
  active_func_ptr := namptr;
  gen('func begin active_func_ptr'); }

1-D Representation of 3-D Array



Code Template for *Expressions and Assignments*

```
int a[10][20][35], b;
b = exp1;
code for evaluation of exp1 (result in T1)
b = T1
/* Assuming the array access to be, a[i][j][k] */
/* base address = addr(a), offset = (((i*n2)+j)*n3)+k)*ele_size */
a[exp2][exp3][exp4] = exp5;
```

```
10: code for exp2 (result in T2)  | | 141: T8 = T7+T6
70: code for exp3 (result in T3)  | | 142: T9 = T8*intsize
105: T4 = T2*20                   | | 143: T10 = addr(a)
106: T5 = T4+T3                   | | 144: code for exp5 (result in T11)
107: code for exp4 (result in T6) | | 186: T10[T9] = T11
140: T7 = T5*35
```

SATG for *Expressions and Assignments*

- $S \rightarrow L := E$

  /* L has two attributes, L.place, pointing to the name of the variable or temporary in the symbol table, and L.offset, pointing to the temporary holding the offset into the array (NULL in the case of a simple variable) */

  { if (L.offset == NULL) gen('L.place = E.result');
  else gen('L.place[L.offset] = E.result');}

- $E \rightarrow (E_1)$ {E.result := E_1.result; }

- $E \rightarrow L$ { if (L.offset == NULL) E.result := L.place;
  else { E.result := newtemp(L.type);
  gen('E.result = L.place[L.offset]'); }

- $E \rightarrow num$ { E.result := newtemp(num.type);
  gen('E.result = num.value'); }

SATG for *Expressions and Assignments* (contd.)

- $E \rightarrow E_1 + E_2$

```
{ result_type := compatible_type(E1.type, E2.type);
  E.result := newtemp(result_type);
  if (E1.type == result_type) operand_1 := E1.result;
  else if (E1.type == integer && result_type == real)
       { operand_1 := newtemp(real);
         gen('operand_1 = cnvrt_float(E1.result); };
  if (E2.type == result_type) operand_2 := E2.result;
  else if (E2.type == integer && result_type == real)
       { operand_2 := newtemp(real);
         gen('operand_2 = cnvrt_float(E2.result); };
  gen('E.result = operand_1 + operand_2');
}
```

SATG for *Expressions and Assignments* (contd.)

- $E \rightarrow E_1 || E_2$
  { E.result := newtemp(integer);
  gen('E.result = E_1.result || E_2.result');
- $E \rightarrow E_1 < E_2$
  { E.result := newtemp(integer);
  gen('E.result = 1');
  gen('if E_1.result < E_2.result goto nextquad+2');
  gen('E.result = 0');
  }
- $L \rightarrow id$ { search_var_param(id.name, active_func_ptr, level, found, vn); L.place := vn; L.offset := NULL; }

Note: *search_var_param*() searches for *id.name* in the variable list first, and if not found, in the parameter list next.

SATG for *Expressions and Assignments* (contd.)

- $ELIST \rightarrow id\ [\ E$
  { search_var_param(id.name, active_func_ptr,
  level, found, vn); ELIST.dim := 1;
  ELIST.arrayptr := vn; ELIST.result := E.result; }
- $L \rightarrow ELIST\ ]$ { L.place := ELIST.arrayptr;
  temp := newtemp(int); L.offset := temp;
  ele_size := ELIST.arrayptr -> ele_size;
  gen('temp = ELIST.result * ele_size'); }
- $ELIST \rightarrow ELIST_1\ ,\ E$
  { ELIST.dim := $ELIST_1$.dim + 1;
  ELIST.arrayptr := $ELIST_1$.arrayptr
  num_elem := get_dim($ELIST_1$.arrayptr, $ELIST_1$.dim + 1);
  temp1 := newtemp(int); temp2 := newtemp(int);
  gen('temp1 = $ELIST_1$.result * num_elem');
  ELIST.result := temp2; gen('temp2 = temp1 + E.result'); }

Short Circuit Evaluation for Boolean Expressions

- (exp1 && exp2): value = if ($\sim$exp1) then FALSE else exp2
  - This implies that exp2 need not be evaluated if exp1 is FALSE
- (exp1 || exp2):value = if (exp1) then TRUE else exp2
  - This implies that exp2 need not be evaluated if exp1 is TRUE
- Since boolean expressions are used mostly in conditional and loop statements, it is possible to realize perform short circuit evaluation of expressions using control flow constructs
- In such a case, there are no explicit '||' and '&&' operators in the intermediate code (as earlier), but only jumps
- Much faster, since complete expression is not evaluated
- If unevaluated expressions have side effects, then program may have non-deterministic behaviour

Control-Flow Realization of Boolean Expressions

if ((a+b < c+d) || ((e==f) && (g > h-k))) A1; else A2; A3;

```
100:          T1 = a+b
101:          T2 = c+d
103:          if T1 < T2 goto L1
104:          goto L2
105:L2:       if e==f goto L3
106:          goto L4
107:L3:       T3 = h-k
108:          if g > T3 goto L5
109:          goto L6
110:L1:L5:    code for A1
111:          goto L7
112:L4:L6:    code for A2
113:L7:       code for A3
```