Intermediate Code Generation - Part 3

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

Short Circuit Evaluation for Boolean Expressions

- (exp1 && exp2): value = if (∼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
```

SATG for *Control-Flow Realization of Boolean Expressions*

- $E \rightarrow E_1 \;||\; M\; E_2$ { backpatch(E_1.falselist, M.quad);
  E.truelist := merge(E_1.truelist, E_2.truelist);
  E.falselist := E_2.falselist }
- $E \rightarrow E_1 \;\&\&\; M\; E_2$ { backpatch(E_1.truelist, M.quad);
  E.falselist := merge(E_1.falselist, E_2.falselist);
  E.truelist := E_2.truelist }
- $E \rightarrow \sim E_1$ { E.truelist := E_1.falselist;
  E.falselist := E_1.truelist }
- $M \rightarrow \epsilon$ {M.quad := nextquad; }
- $E \rightarrow E_1 < E_2$ { E.truelist := makelist(nextquad);
  E.falselist := makelist(nextquad+1);
  gen('if E_1.result < E_2.result goto __');
  gen('goto __'); }

SATG for *Control-Flow Realization of Boolean Expressions*

- $E \rightarrow (E_1)$
  { E.truelist := E_1.truelist; E.falselist := E_1.falselist }
- $E \rightarrow$ *true* { E.truelist := makelist(nextquad); gen('goto __');}
- $E \rightarrow$ *false*
  { E.falselist := makelist(nextquad); gen('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); }
- $IFEXP \rightarrow if\ \ E$ { backpatch(E.truelist, nextquad);
  IFEXP.falselist := E.falselist;}
- $N \rightarrow \epsilon$ { N.next := makelist(nextquad); gen('goto __'); }

SATG for *Control-Flow Realization of Boolean Expressions*

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

Code Template for *Switch* Statement

switch (exp) {
  case l_1 : SL_1
  case l_{2_1}: case l_{2_2} : SL_2
  ...
  case l_{n-1} : SL_{n-1}
  default: SL_n
}

This code template can be used for switch statements with 10-15 cases. Note that statement list SL_i must incorporate a 'break' statement, if necessary

	code for exp (result in T)
	goto TEST
L_1:	code for SL_1
L_2:	code for SL_2
	...
L_n:	code for SL_n
	goto NEXT
TEST:	if T==l_1 goto L_1
	if T==l_{2_1} goto L_2
	if T==l_{2_2} goto L_2
	...
	if T==l_{n-1} goto L_{n-1}
	if default_yes goto L_n
NEXT:	

Grammar for Switch Statement

The grammar for the 'switch' statement according to ANSI standard C is:

selection_statement → SWITCH '(' expression ')' statement

However, a more intuitive form of the grammar is shown below

- *STMT* → *SWITCH_HEAD SWITCH_BODY*
- *SWITCH_HEAD* → *switch* (*E*)/* E must be `int` type */
- *SWITCH_BODY* → { *CASE_LIST* }
- *CASE_LIST* → *CASE_ST* | *CASE_LIST CASE_ST*
- *CASE_ST* → *CASE_LABELS STMT_LIST* ;
- *CASE_LABELS* → ϵ | *CASE_LABELS CASE_LABEL*
- *CASE_LABEL* → *case CONST_INTEXPR* : | *default* :
  /* CONST_INTEXPR must be of `int` or `char` type */
- *STMT* → *break* /* also an option */

SATG for *Switch* Statement

- *SWITCH_HEAD* $\rightarrow$ *switch (E)*
  { SWITCH_HEAD.result := E.result;
  SWITCH_HEAD.test := nextquad;
  gen(‘goto __’); }
- *STMT* $\rightarrow$ *break*
  { STMT.next := makelist(nextquad);
  gen(‘goto __’); }
- *CASE_LABEL* $\rightarrow$ *case CONST_INTEXPR* :
  { CASE_LABEL.val := CONST_INTEXPR.val;
  CASE_LABEL.default := false; }
- *CASE_LABEL* $\rightarrow$ *default* : {CASE_LABEL.default := true; }
- *CASE_LABELS* $\rightarrow$ ϵ { CASE_LABELS.default := false;
  { CASE_LABELS.list := makelist(NULL); }

SATG for *Switch* Statement (contd.)

- $CASE\_LABELS \rightarrow CASE\_LABELS_1 \quad CASE\_LABEL$
  { if (∼CASE_LABEL.default) CASE_LABELS.list :=
  append($CASE\_LABELS_1$.list, CASE_LABEL.val);
  else CASE_LABELS.list := $CASE\_LABELS_1$.list;
  if ($CASE\_LABELS_1$.default || CASE_LABEL.default)
  CASE_LABEL.default := true; }
- $CASE\_ST \rightarrow CASE\_LABELS \quad M \quad STMT\_LIST\ ;$
  { CASE_ST.next := STMT_LIST.next; CASE_ST.list :=
  add_jump_target(CASE_LABELS.list, M.quad);
  if (CASE_LABELS.default) CASE_ST.default := M.quad;
  else CASE_ST.default := -1; }
- $CASE\_LIST \rightarrow CASE\_ST$
  { CASE_LIST.next := CASE_ST.next;
  CASE_LIST.list := CASE_ST.list;
  CASE_LIST.default := CASE_ST.default; }

Code Template for *Switch* Statement

switch (exp) {
  case l_1 : SL_1
  case l_{2_1} : case l_{2_2} : SL_2
  ...
  case l_{n-1} : SL_{n-1}
  default: SL_n
}

This code template can be used for switch statements with 10-15 cases. Note that statement list SL_i must incorporate a 'break' statement, if necessary

	code for exp (result in T)
	goto TEST
L_1:	code for SL_1
L_2:	code for SL_2
	...
L_n:	code for SL_n
	goto NEXT
TEST:	if T==l_1 goto L_1
	if T==l_{2_1} goto L_2
	if T==l_{2_2} goto L_2
	...
	if T==l_{n-1} goto L_{n-1}
	if default_yes goto L_n
NEXT:	

SATG for *Switch* Statement (contd.)

- *CASE_LIST* $\rightarrow$ *CASE_LIST*$_1$ *CASE_ST*
  { CASE_LIST.next :=
  merge(*CASE_LIST*$_1$.next, CASE_ST.next);
  CASE_LIST.list :=
  merge(*CASE_LIST*$_1$.list, CASE_ST.list);
  CASE_LIST.default := *CASE_LIST*$_1$.default == -1 ?
  CASE_ST.default : *CASE_LIST*$_1$.default; }
- *SWITCH_BODY* $\rightarrow$ { *CASE_LIST* }
  { SWITCH_BODY.next :=
  merge(CASE_LIST.next, makelist(nextquad));
  gen('goto __');
  SWITCH_BODY.list := CASE_LIST.list;
  SWITCH_BODY.default := CASE_LIST.default; }

SATG for *Switch* Statement (contd.)

- *STMT → SWITCH_HEAD SWITCH_BODY*

```
{ backpatch(SWITCH_HEAD.test, nextquad);
  for each (value, jump) pair in SWITCH_BODY.list do {
    (v,j) := next (value, jump) pair from SWITCH_BODY.list;
    gen('if SWITCH_HEAD.result == v goto j');
  }
  if (SWITCH_BODY.default != -1)
      gen('goto SWITCH_BODY.default');
  STMT.next := SWITCH_BODY.next;
}
```

C For-Loop

The for-loop of C is very general

- for (*expression*$_1$; *expression*$_2$; *expression*$_3$) *statement*
  This statement is equivalent to
  expression$_1$;
  while (*expression*$_2$) { *statement* *expression*$_3$; }
- All three expressions are optional and any one (or all) may be missing
- Code generation is non-trivial because the order of execution of *statement* and *expression*$_3$ are reversed compared to their occurrance in the for-statement
- Difficulty is due to 1-pass bottom-up code generation
- Code generation during parse tree traversals mitigates this problem by generating code for *expression*$_3$ before that of *statement*

Code Generation Template for *C For-Loop*

for (E_1; E_2; E_3) S

```
        code for E1
L1:     code for E2 (result in T)
        goto L4
L2:     code for E3
        goto L1
L3:     code for S /* all jumps out of S goto L2 */
        goto L2
L4:     if  T == 0  goto  L5 /* if T is zero, jump to exit */
        goto L3
L5:     /* exit */
```

Code Generation for C For-Loop

- $STMT \rightarrow for\ (\ E_1;\ M\ E_2;\ N\ E_3\ )\ P\ STMT_1$
  { gen('goto N.quad+1'); Q1 := nextquad;
  gen('if E_2.result == 0 goto __');
  gen('goto P.quad+1');
  backpatch(N.quad, Q1);
  backpatch($STMT_1$.next, N.quad+1);
  backpatch(P.quad, M.quad);
  STMT.next := makelist(Q1); }
- $M \rightarrow \epsilon$ { M.quad := nextquad; }
- $N \rightarrow \epsilon$ { N.quad := nextquad; gen('goto __'); }
- $P \rightarrow \epsilon$ { P.quad := nextquad; gen('goto __'); }

ALGOL For-Loop

- Let us also consider a more restricted form of the for-loop
  - $STMT \rightarrow for\ id = EXP_1\ to\ EXP_2\ by\ EXP_3\ do\ STMT_1$
    where, EXP_1, EXP_2, and EXP_3 are all arithmetic expressions, indicating starting, ending and increment values of the iteration index
  - EXP_3 may have either positive or negative values
  - All three expressions are evaluated before the iterations begin and are stored. They are not evaluated again during the loop-run
  - All three expressions are mandatory (unlike in the C-for-loop)

Code Generation Template for ALGOL For-Loop

$STMT \rightarrow for\ id = EXP_1\ to\ EXP_2\ by\ EXP_3\ do\ STMT_1$

```
      Code for EXP1 (result in T1)
      Code for EXP2 (result in T2)
      Code for EXP3 (result in T3)
      goto L1
L0:   Code for STMT1
      id = id + T3
      goto L2
L1:   id = T1
L2:   if (T3 ≤ 0) goto L3
      if (id > T2) goto L4 /* positive increment */
      goto L0
L3:   if (id < T2) goto L4 /* negative increment */
      goto L0
L4:
```

Code Generation for ALGOL For-Loop

$M \rightarrow \epsilon$ { M.quad := nextquad; gen('goto __'); }

$STMT \rightarrow$ *for id* $=$ EXP_1 *to* EXP_2 *by* EXP_3 *M do* $STMT_1$
{ search(id.name, idptr); gen('idptr = idptr + EXP_3.result');
Q1 := nextquad; gen('goto __'); backpatch(M.quad, nextquad);
gen('idptr = EXP_1.result'); backpatch(Q1, nextquad);
Q2 := nextquad; gen('if EXP_3.result $\leq$ 0 goto __');
gen('if idptr > EXP_2.result goto __');
gen('goto M.quad+1'); backpatch(Q2, nextquad);
Q3 := nextquad; gen('if idptr < EXP_2.result goto __');
gen('goto M.quad+1');
STMT.next :=
merge(makelist(Q2+1), makelist(Q3), $STMT_1$.next);

Another Code Generation Template for ALGOL For-Loop

$STMT \rightarrow for\ id = EXP_1\ to\ EXP_2\ by\ EXP_3\ do\ STMT_1$

```
      Code for EXP1 (result in T1)
      Code for EXP2 (result in T2)
      Code for EXP3 (result in T3)
      id = T1
L1:   if (T3 ≤ 0) goto L2
      if (id > T2) goto L4 /* positive increment */
      goto L3
L2:   if (id < T2) goto L4 /* negative increment */
L3:   Code for STMT
      id = id + T3
      goto L1
L4:
```

Code generation using this template is left as an exercise

Run-Time Array Range Checking

`int b[10][20]; a = b[`exp_1`][`exp_2`];`

The code generated for this assignment with run-time array range checking is as below:

```
     code for exp1 /* result in T1 */
     if T1 < 10 goto L1
     'error: array overflow in dimension 1'
     T1 = 9 /* max value for dim 1 */
L1:  code for exp2 /* result in T2 */
     if T2 < 20 goto L2
     'error: array overflow in dimension 2'
     T2 = 19 /* max value for dim 2 */
L2:  T3 = T1*20
     T4 = T3+T2
     T5 = T4*intsize
     T6 = addr(b)
     a = T6[T5]
```

Code Generation with Array Range Checking

- $S \rightarrow L \ := \ E$
  { if (L.offset == NULL) gen('L.place = E.result');
  else gen('L.place[L.offset] = E.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]'); }
- $ELIST \rightarrow id \ [\ E$ { search_var(id.name, active_func_ptr,
  level, found, vn); ELIST.arrayptr := vn;
  ELIST.result := E.result; ELIST.dim := 1;
  num_elem := get_dim(vn, 1); Q1 := nextquad;
  gen('if E.result < num_elem goto Q1+3');
  gen('error("array overflow in dimension 1")');
  gen('E.result = num_elem-1');}

Code Generation with Array Range Checking(contd.)

- $L \rightarrow ELIST$] { 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);
  Q1 := nextquad;
  gen('if E.result < num_elem goto Q1+3');
  gen('error("array overflow in ($ELIST_1$.dim + 1)")');
  gen('E.result = num_elem-1');
  temp1 := newtemp(int); temp2 := newtemp(int);
  gen('temp1 = $ELIST_1$.result * num_elem');
  ELIST.result := temp2; gen('temp2 = temp1 + E.result'); }