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 ($\sim$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
```

Code Template for *Switch* Statement

```
switch (exp) {
    case I1 : SL1
    case I21 : case I22 : SL2
    ...
    case In−1 : SLn−1
    default: SLn
}
```

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
L1:     code for SL1
L2:     code for SL2
        ...
Ln:     code for SLn
        goto NEXT
TEST:   if T==I1 goto L1
        if T==I21 goto L2
        if T==I22 goto L2
        ...
        if T==In−1 goto Ln−1
        if default_yes goto Ln
NEXT:
```

Grammar for Switch Statement

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

selection_statement $\rightarrow$ SWITCH '(' expression ')' statement

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

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

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 */
```

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:
```