Intermediate Code Generation - Part 4

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

break and *continue* Statements

- `break` statements can occur only within `while, for, do-while` and `switch` statements
- `continue` statements can occur only within `while, for,` and `do-while` statements (i.e., only loops)
- All other occurrences are flagged as errors by the compiler
- Examples (incorrect programs)
  - ```
    main(){
      int a=5;
      if (a<5) {break; printf("hello-1");};
      printf("hello-2");}
    }
    ```
  - Replacing `break` with `continue` in the above program is also erroneous
    ```

break and *continue* Statements (correct programs)

- The program below prints 6

```
main(){int a,b=10; for(a=1;a<5;a++) b--;
        printf("%d",b);}
```

- The program below prints 8

```
main(){int a,b=10; for(a=1;a<5;a++)
   { if (a==3) break; b--;} printf("%d",b);}
```

- The program below prints 7

```
main(){int a,b=10; for(a=1;a<5;a++)
   { if (a==3) continue; b--;} printf("%d",b);}
```

- This program also prints 8

```
main(){int a,b=10; for(a=1;a<5;a++)
   { while (1) break;
     if (a==3) break; b--;} printf("%d",b);}
```

Handling *break* and *continue* Statements

- We need extra attributes for the non-terminal *STMT*
  - *STMT.break* and *STMT.continue*, along with *STMT.next*(existing one), all of which are lists of quadruples with unfilled branch targets
- *STMT* $\rightarrow$ *break*
  { STMT.break := makelist(nextquad); gen('goto __');
  STMT.next := makelist(NULL);
  STMT.continue := makelist(NULL); }
- *STMT* $\rightarrow$ *continue*
  { STMT.continue := makelist(nextquad); gen('goto __');
  STMT.next := makelist(NULL);
  STMT.break := makelist(NULL); }

SATG for *While-do* Statement with *break* and *continue*

- $WHILEXEP \rightarrow while \;\; M \;\; E$
  { WHILEEXP.falselist := makelist(nextquad);
  gen('if E.result $\leq$ 0 goto __');
  WHILEEXP.begin := M.quad; }
- $STMT \rightarrow WHILEXEP \;\; do \;\; STMT_1$
  { gen('goto WHILEEXP.begin');
  backpatch($STMT_1$.next, WHILEEXP.begin);
  backpatch($STMT_1$.continue, WHILEEXP.begin);
  STMT.continue := makelist(NULL);
  STMT.break := makelist(NULL);
  STMT.next := merge(WHILEEXP.falselist, $STMT_1$.break); }
- $M \rightarrow \epsilon$
  { M.quad := nextquad; }

Code Generation Template for *C For-Loop* with *break* and *continue*

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 breaks out of S goto L5 */
/* all continues and other 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 with *break* and *continue*

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

LATG for *If-Then-Else* Statement

Assumption: No short-circuit evaluation for E

If (E) S1 else S2

```
        code for E (result in T)
        if T≤ 0 goto L1 /* if T is false, jump to else part */
        code for S1 /* all exits from within S1 also jump to L2 */
        goto L2 /* jump to exit */
L1:     code for S2 /* all exits from within S2 also jump to L2 */
L2:     /* exit */
```

$S \rightarrow$ *if E* { N := nextquad; gen('if E.result <= 0 goto __'); }
S_1 *else* { M := nextquad; gen('goto __');
backpatch(N, nextquad); }
S_2 { S.next := merge(makelist(M), S_1.next, S_2.next); }

LATG for *While-do* Statement

Assumption: No short-circuit evaluation for E

```
while  (E)  do  S
L1:      code for E (result in T)
         if  T≤ 0  goto  L2 /* if T is false, jump to exit */
         code for S /* all exits from within S also jump to L1 */
         goto L1 /* loop back */
L2:      /* exit */
```

$S \rightarrow$ *while* { M := nextquad; }
E { N := nextquad; gen('if E.result <= 0 goto __'); }
do S_1 { backpatch(S_1.next, M); gen('goto M');
S.next := makelist(N); }

LATG for Other Statements

- $S \rightarrow A$ { S.next := makelist(NULL); }
- $S \rightarrow \{\ SL\ \}$ { S.next := SL.next; }
- $SL \rightarrow \epsilon$ { SL.next := makelist(NULL); }
- $SL \rightarrow S;$ { backpatch(S.next, nextquad); }
  SL_1 { SL.next := SL_1.next; }
- When a function ends, we perform { gen('func end'); }. No backpatching of SL.next is required now, since this list will be empty, due to the use of $SL \rightarrow \epsilon$ as the last production.
- LATG for function declaration and call, and return statement are left as exercises

LATG for Expressions

- $A \to L = E$
  { if (L.offset == NULL) /* simple id */
  gen('L.place = E.result');
  else gen('L.place[L.offset] = E.result'); }
- $E \to T$ { E'.left := T.result; }
  E' { E.result := E'.result; }
- $E' \to + T$ { temp := newtemp(T.type);
  gen('temp = E'.left + T.result'); E'_1.left := temp; }
  E'_1 { E'.result := E'_1.result; }
  Note: Checking for compatible types, etc., are all required here as well. These are left as exercises.
- $E' \to \epsilon$ { E'.result := E'.left; }
- Processing $T \to F\ T'$, $T' \to *F\ T' \mid \epsilon$, $F \to (\ E\ )$, boolean and relational expressions are all similar to the above productions

LATG for Expressions(contd.)

- $F \rightarrow L$ { if (L.offset == NULL) F.result := L.place;
  else { F.result := newtemp(L.type);
  gen('F.result = L.place[L.offset]'); }
- $F \rightarrow num$ { F.result := newtemp(num.type);
  gen('F.result = num.value'); }
- $L \rightarrow id$ { search(id.name, vn); INDEX.arrayptr := vn; }
  $INDEX$ { L.place := vn; L.offset := INDEX.offset; }
- $INDEX \rightarrow \epsilon$ { INDEX.offset := NULL; }
- $INDEX \rightarrow$ [{ ELIST.dim := 1;
  ELIST.arrayptr := INDEX.arrayptr; }
  $ELIST$]
  { temp := newtemp(int); INDEX.offset := temp;
  ele_size := INDEX.arrayptr -> ele_size;
  gen('temp = ELIST.result * ele_size'); }

LATG for Expressions(contd.)

- *ELIST* $\rightarrow$ *E* { INDEXLIST.dim := ELIST.dim+1;
  INDEXLIST.arrayptr := ELIST.arrayptr;
  INDEXLIST.left := E.result; }
  INDEXLIST { ELIST.result := INDEXLIST.result; }
- *INDEXLIST* $\rightarrow \epsilon$ { INDEXLIST.result := INDEXLIST.left; }
- *INDEXLIST* $\rightarrow$ **,** { **action 1** }
  ELIST { gen('temp = temp + ELIST.result');
  INDEXLIST.result := temp; }

action 1:

{ temp := newtemp(int);
num_elem := rem_num_elem(INDEXLIST.arrayptr,
INDEXLIST.dim);
gen('temp = INDEXLIST.left * num_elem');
ELIST.arrayptr := INDEXLIST.arrayptr;
ELIST.dim := INDEXLIST.dim; }

LATG for Expressions(contd.)

- The function rem_num_elem(arrayptr, dim) computes the product of the dimensions of the array, starting from dimension *dim*. For example, consider the expression, `a[i,j,k,l]`, and its declaration `int a[10,20,30,40]`. The expression translates to $i * 20 * 30 * 40 + j * 30 * 40 + k * 40 + l$. The above function returns, 24000(dim=2), 1200(dim=3), and 40(dim=3).

Run-time Environments - 1

Y.N. Srikant
Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Outline of the Lecture

- What is run-time support?
- Parameter passing methods
- Storage allocation
- Activation records
- Static scope and dynamic scope
- Passing functions as parameters
- Heap memory management
- Garbage Collection

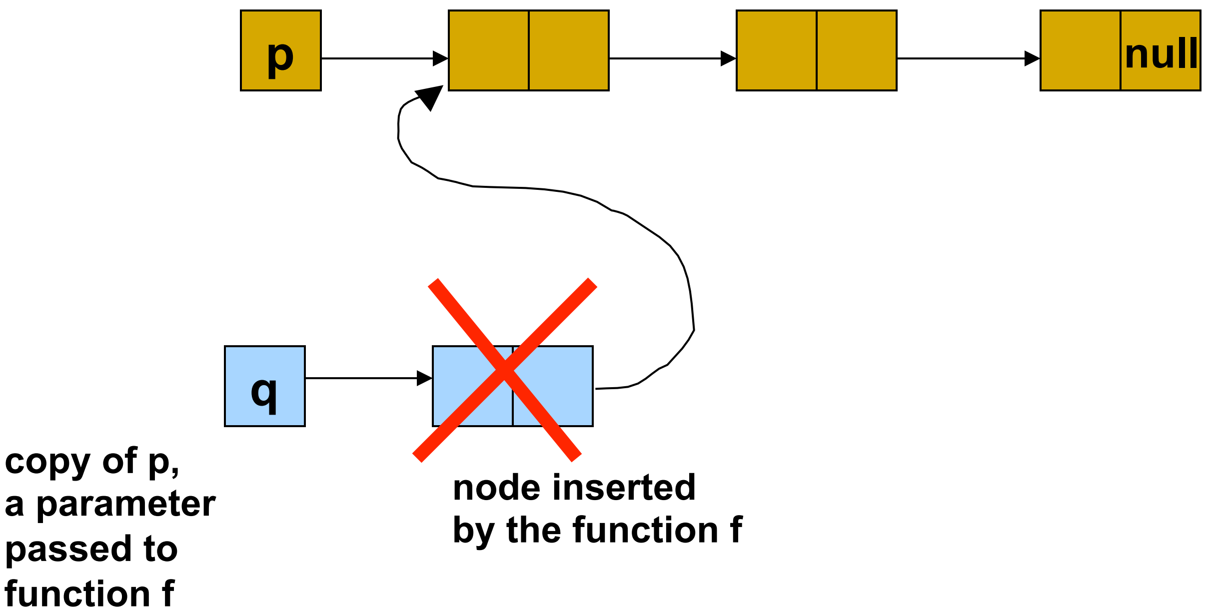
What is Run-time Support?

- It is not enough if we generate machine code from intermediate code
- Interfaces between the program and computer system resources are needed
  - There is a need to manage memory when a program is running
    - This memory management must connect to the data objects of programs
    - Programs request for memory blocks and release memory blocks
    - Passing parameters to fucntions needs attention
  - Other resources such as printers, file systems, etc., also need to be accessed
- These are the main tasks of run-time support
- In this lecture, we focus on memory management

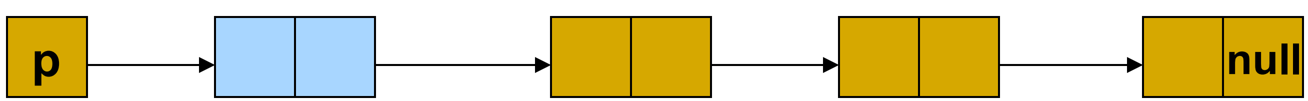
Parameter Passing Methods - Call-by-value

- At runtime, prior to the call, the parameter is evaluated, and its actual value is put in a location private to the called procedure
  - Thus, there is no way to change the actual parameters.
  - Found in C and C++
  - C has only call-by-value method available
    - Passing pointers does not constitute call-by-reference
    - Pointers are also copied to another location
    - Hence in C, there is no way to write a function to insert a node at the front of a linked list (just after the header) without using pointers to pointers

Problem with Call-by-Value



node insertion as desired



Parameter Passing Methods - Call-by-Reference

- At runtime, prior to the call, the parameter is evaluated and put in a temporary location, if it is not a variable
- The **address** of the variable (or the temporary) is passed to the called procedure
- Thus, the actual parameter may get changed due to changes to the parameter in the called procedure
- Found in C++ and Java

Call-by-Value-Result

- ***Call-by-value-result*** is a hybrid of Call-by-value and Call-by-reference
- Actual parameter is calculated by the calling procedure and is copied to a local location of the called procedure
- Actual parameter's value is not affected during execution of the called procedure
- At return, the value of the formal parameter is copied to the actual parameter, if the actual parameter is a variable
- Becomes different from call-by-reference method
  - when global variables are passed as parameters to the called procedure and
  - the same global variables are also updated in another procedure invoked by the called procedure
- Found in Ada

Difference between Call-by-Value, Call-by-Reference, and Call-by-Value-Result

```
int a;
void Q()
    { a = a+1; }
void R(int x);
    { x = x+10; Q(); }
main()
    { a = 1; R(a); print(a); }
```

call-by-value	call-by-reference	call-by-value-result
2	**12**	**11**

Value of a printed

Note: In Call-by-V-R, value of x is copied into a, when proc R returns. Hence a=11.

Parameter Passing Methods - Call-by-Name

- Use of a call-by-name parameter implies a **textual** substitution of the formal parameter name by the **actual** parameter
- For example, if the procedure

  void R (int X, int I);

  { I = 2; X = 5; I = 3; X = 1; }

  is called by R(B[J*2], J)

  this would result in (effectively) changing the body to

  { J = 2; B[J*2] = 5; J = 3; B[J*2] = 1; }

  just before executing it

Parameter Passing Methods - Call by Name

- Note that the actual parameter corresponding to *X* changes whenever *J* changes
  - Hence, we cannot evaluate the address of the actual parameter just once and use it
  - It must be recomputed every time we reference the formal parameter within the procedure
- A separate routine (called *thunk*) is used to evaluate the parameters whenever they are used
- Found in Algol and functional languages

Example of Using the Four Parameter Passing Methods

```
void swap (int x, int y)
{ int temp;
  temp = x;
  x = y;
  y = temp;
} /*swap*/
...
{ i = 1;
  a[i] =10; /* int a[5]; */
  print(i,a[i]);
  swap(i,a[i]);
  print(i,a[1]); }
```

- Results from the 4 parameter passing methods (print statements)

call-by-value	call-by-reference	call-by-val-result	call-by-name
1 10	1 10	1 10	1 10
1 10	10 1	10 1	error!

Reason for the error in the Call-by-name Example

The problem is in the swap routine

temp = i; /* => temp = 1 */
i = a[i]; /* => i =10 since a[i] ==10 */
a[i] = temp; /* => a[10] = 1 => index out of bounds */