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);}
```

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 */
```

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

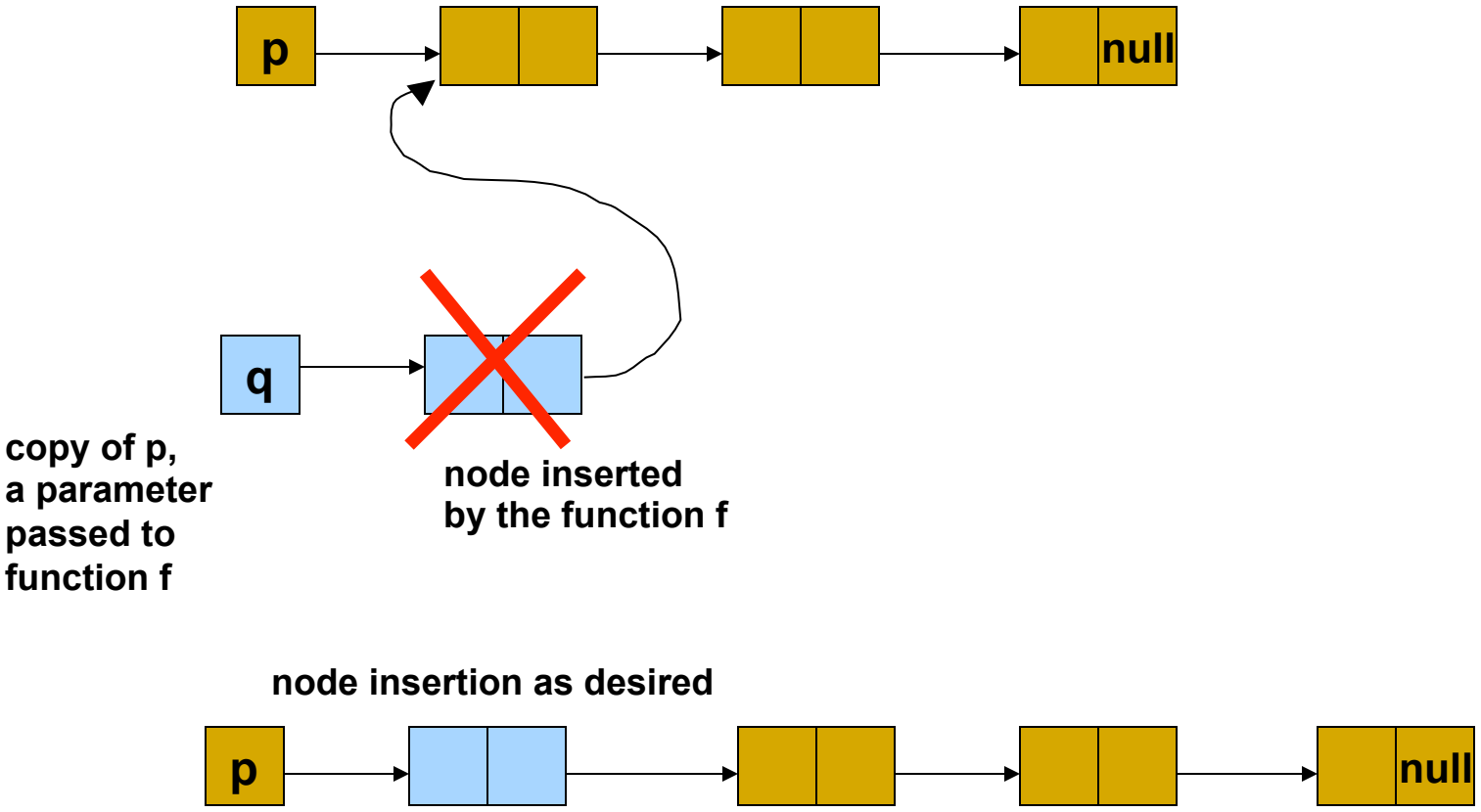
What is Run-time Support?

- 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



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

```
1.  void swap (int x, int y)
2.  { int temp;
3.    temp = x;
4.    x = y;
5.    y = temp;
6.  } /*swap*/
7.  ...
8.  { i = 1;
9.    a[i] =10; /* int a[5]; */
10.   print(i,a[i]);
11.   swap(i,a[i]);
12.   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 10 1	1 10 10 1	1 10 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 */