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)
  ```c
  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

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

- The program below prints 8

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

- The program below prints 7

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

- This program also prints 8

```c
main() {int a, b=10; for(a=1; a<5; a++)
    { while (1) break;
        if (a==3) break; b--; } printf("%d", b);
}
```
Handling \textit{break} and \textit{continue} Statements

- We need extra attributes for the non-terminal \textit{STMT}
  - \textit{STMT.break} and \textit{STMT.continue}, along with \textit{STMT.next}(existing one), all of which are lists of quadruples with unfilled branch targets

\begin{itemize}
  \item \textit{STMT} \rightarrow \textit{break}\n    \{ STMT.break := \text{makelist}(\text{nextquad}); \text{gen(‘goto __’)}; \text{STMT.next} := \text{makelist}(\text{NULL}); \text{STMT.continue} := \text{makelist}(\text{NULL}); \} \n  \item \textit{STMT} \rightarrow \textit{continue}\n    \{ STMT.continue := \text{makelist}(\text{nextquad}); \text{gen(‘goto __’)}; \text{STMT.next} := \text{makelist}(\text{NULL}); \text{STMT.break} := \text{makelist}(\text{NULL}); \} \n\end{itemize}
SATG for *While-do* Statement with *break* and *continue*

- **WHILEEXP** $\rightarrow$ *while M E*
  
  
  ```
  { WHILEEXP.falselist := makelist(nextquad);
    gen('if E.result $\leq$ 0 goto ___');
    WHILEEXP.begin := M.quad; }
  ```

- **STMT** $\rightarrow$ **WHILEEXP** *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; }
  ```
for ( \( E_1; \ E_2; \ E_3 \) ) S
  code for \( E_1 \)
L1: code for \( E_2 \) (result in T)
goto L4
L2: code for \( E_3 \)
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 \textit{break} and \textit{continue}

\begin{itemize}
  \item \textbf{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); \}
  \item \textbf{M} \rightarrow \epsilon \{ M.quad := nextquad; \}
  \item \textbf{N} \rightarrow \epsilon \{ N.quad := nextquad; gen('goto ___'); \}
  \item \textbf{P} \rightarrow \epsilon \{ P.quad := nextquad; gen('goto ___'); \}
\end{itemize}
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 \( T \leq 0 \) 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 */

\[ S \rightarrow \text{if } E \{ \text{N := nextquad; gen('if E.result }\leq 0 \text{ goto __')}; \} \]
\[ S_1 \text{ else } \{ \text{M := nextquad; gen('goto __')}; \]
\[ \text{backpatch(N, nextquad); } \}
\[ S_2 \{ \text{S.next := merge(makelist(M), } S_1.\text{next, } S_2.\text{next}); } \]
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 → while { M := nextquad; }
    E { N := nextquad; gen('if E.result <= 0 goto ___'); }
    do S₁ { backpatch(S₁.next, M); gen('goto M');
        S.next := makelist(N); }

Y.N. Srikant Intermediate Code Generation
LATG for Other Statements

- $S \rightarrow A \{ \text{S.next := makelist(NULL); } \}$
- $S \rightarrow \{ SL \} \{ \text{S.next := SL.next; } \}$
- $SL \rightarrow \epsilon \{ \text{SL.next := makelist(NULL); } \}$
- $SL \rightarrow S; \{ \text{backpatch(S.next, nextquad); } \}$
  $SL_1 \{ \text{SL.next := SL}_1.\text{next; } \}$

When a function ends, we perform $\{ \text{gen(‘func end’); } \}$. No backpatching of $\text{SL.next}$ 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 → L = E**
  
  { if (L.offset == NULL) /* simple id */
    \( \text{gen('L.place = E.result');} \)
  
  else \( \text{gen('L.place[L.offset] = E.result');} \) }

- **E → T** { E’.left := T.result; }
  
  **E’** { E.result := E’.result; }

- **E’ → + T** { temp := newtemp(T.type);
  
  \( \text{gen('temp = E'.left + T.result');} \)
  
  \( E'_1 \) { E’.result := E’1.result; }

  Note: Checking for compatible types, etc., are all required here as well. These are left as exercises.

- **E’ → ϵ** { E’.result := E’.left; }

- Processing **T → F T’, T’ → *F T’ | ϵ**, **F → ( 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 {
      gen('F.result = L.place[L.offset]');
      F.result := newtemp(L.type);
  }
  ```

- **$F \rightarrow num$**: 
  ```
  gen('F.result = num.value');
  ```

- **$L \rightarrow id$**: 
  ```
  search(id.name, vn); INDEX.arrayptr := vn;
  ```

- **INDEX**: 
  ```
  L.place := vn; L.offset := INDEX.offset;
  ```

- **INDEX**
  ```
  INDEX.offset := NULL;
  ```

- **INDEX**
  ```
  ELIST.dim := 1;
  ELIST.arrayptr := INDEX.arrayptr;
  ```

- **ELIST**
  ```
  gen('temp = ELIST.result * ele_size');
  ```

Y.N. Srikant  Intermediate Code Generation
LATG for Expressions (contd.)

- $ELIST \rightarrow E \{ \text{INDEXLIST.dim := ELIST.dim+1};$
  \hspace{1em} INDEXLIST.arrayptr := ELIST.arrayptr;
  \hspace{1em} INDEXLIST.left := E.result; \}$$\text{INDEXLIST} \{ \text{ELIST.result := INDEXLIST.result; } \}$
- $INDEXLIST \rightarrow \epsilon \{ \text{INDEXLIST.result := INDEXLIST.left; } \}$
- $INDEXLIST \rightarrow , \{ \text{action 1} \}$
  \hspace{2em} $ELIST \{ \text{gen('temp = temp + ELIST.result');}$
  \hspace{2em} 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; \}
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

\[ i \times 20 \times 30 \times 40 + j \times 30 \times 40 + k \times 40 + l. \]

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 functions 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

Copy of p, a parameter passed to function f

Node inserted by the function f

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); }

<table>
<thead>
<tr>
<th>call-by-value</th>
<th>call-by-reference</th>
<th>call-by-value-result</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>12</td>
<td>11</td>
</tr>
</tbody>
</table>

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)

<table>
<thead>
<tr>
<th>call-by-value</th>
<th>call-by-reference</th>
<th>call-by-val-result</th>
<th>call-by-name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 10</td>
<td>1 10</td>
<td>1 10</td>
<td>1 10</td>
</tr>
<tr>
<td>1 10</td>
<td>10 1</td>
<td>10 1</td>
<td>error!</td>
</tr>
</tbody>
</table>

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 */
```