Chapter 8
Code Generation

2022 Spring&Summer

Chapter 8 Code Generation

- The characteristics of the source language.
- Information about the target architecture.
- The structure of the runtime environment.
- The operating system running on the target machine.
- A compiler typically breaks up this phase into several steps, involving various intermediate data structures, often including some form of abstract code called intermediate code.

Main issues will be discussed

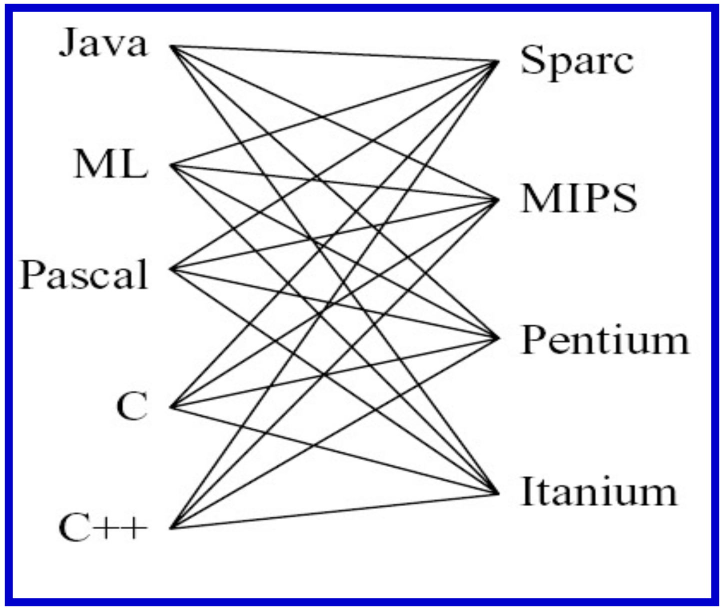
- 1. Two popular forms of intermediate code(three address code and P-code)
- 2. The basic algorithms for generating intermediate or assembly code.
- 3. Code generation techniques for various language features are discussed.
- 4. Apply the techniques studied in previous sections to develop an assembly code generator for the TINY language.

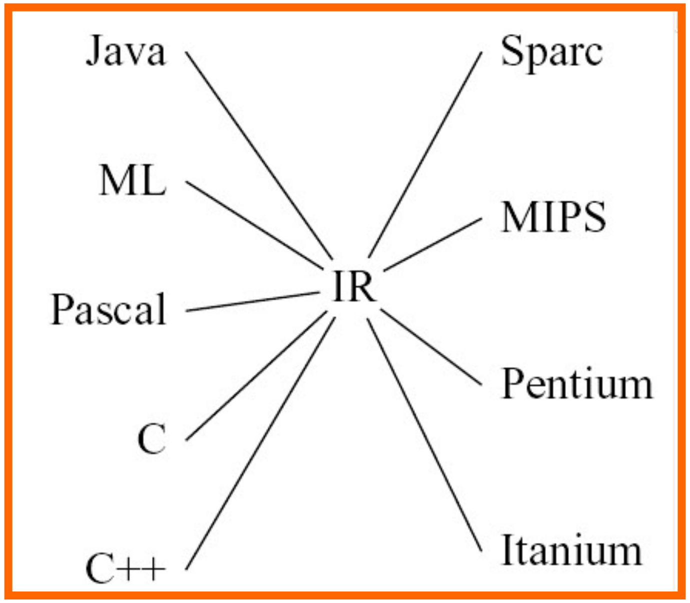
8.1 Intermediate Code and Data Structures for Code Generation

- A data structure that represents the source program during translation is called an intermediate representation, or IR, for short.
- Such an intermediate representation that resembles target code is called intermediate code.
- Intermediate code can be very high level, representing all operations almost as abstractly as a syntax tree, or it can closely resemble target code.
- Intermediate code can also be useful in making a compiler more easily retargetable

8.1 Intermediate Code and Data Structures for Code Generation

- **Translating directly to real machine code**
  - **hinders portability and modularity.**





8.1.1 Three-Address Code

- The most basic instruction of three address code is designed to represent the evaluation of arithmetic expressions and has the following general form:

  x = y op z

  *2*a+(b-3)*

- the corresponding three-address code is as follow

  *T1 = 2 * a*

  T2 = b – 3

  T3 = T1 + T2

- T1、T2 and T3 are temporaries. they will be assigned to registers, they may also be kept in activation records.

Example

Figure 8.1 Sample TINY program

```
{ sample program
   in TINY language --
computes factorial
}
read x ;  { input an integer }
if 0 < x then { don't compute if x <= 0 }
   fact:=1;
   repeat
     fact:=fact*x;
    x:=x-1
until x=0;
write fact { output factorial of x }
    end
```

Figure 8.2 Three-address code

```
read x
t1=x>0
if_false t1 goto L1
fact=1
label   L2
t2 = fact * x
fact = t2
t3 = x - 1
x = t3
t4= x= =0
if_false t4 goto  L2
        write fact
label   L1
halt
```

8.1.2 Data Structures for the Implementation of Three-Address Code

- Four fields are necessary: one for the operation and three for the addresses. Such a representation of three-address code is called a quadruple.
- Those instructions that need fewer than three addresses, one or more of the address fields is given a null or “empty” value.
- The entire sequence of three-address instructions is implemented as an array or linked list.

8.1.2 Data Structures for the Implementation of Three-Address Code

- We allow an address to be only an integer constant or a string (representing the name of a temporary or a variable).
- since names are used, these names must be entered into a symbol table, and lookups will need to be performed during further processing.
- An alternative to keeping names in the **quadruples** is to keep pointers to symbol table entries. This avoids the need for additional lookups and is particularly advantageous in a language with nested scopes .
- A different implementation of three-address code is use the instructions themselves to represent *the temporaries*. Such an implementation of three-address code is called a **triple.**

8.1.2 Data Structures for the Implementation of Three-Address Code

- Quadruple implementation for the three-address code of Figure 8.2.
- (rd, x , _ , _)
- (gt, x, 0, t1)
- (if_f, t1, L1, _)
- (asn, 1,fact, _)
- (lab, L2, _ , _)
- (mul, fact, x, t2)
- (asn, t2, fact, _)
- (sub, x, 1, t3)
- (asn, t3, x, _)
- (eq, x, 0, t4)
- (if_f, t4, L2, _)
- (wri, fact, _, _)
- (lab, L1, _ , _)
- (halt, _, _, _)

Figure 8.2 Three-address code

```
read x
t1=x>0
if_false t1 goto L1
fact=1
label   L2
t2 = fact * x
fact = t2
t3 = x - 1
x = t3
t4= x= =0
if_false t4 goto  L2
          write fact
label   L1
halt
```

8.1.2 Data Structures for the Implementation of Three-Address Code

- Typedef enum { rd, gt, if_f, asn, lab, mul, sub, eq, wri, halt, …} OpKind;
- Typedef enum { Empty, IntConst, String } AddrKind;
- Typedef struct
-         { AddrKind kind;
-                 Union
-                     { int val;
-                       char * name;
-                     } contents;
-          } Address;
- Typedef struct
-         { OpKind op;
-                 Address addr1, addr2,addr3;
-          } Quad

8.1.2 Data Structures for the Implementation of Three-Address Code

Three-address code of Figure 8.2.

```
(0) (rd, x , _  )
(1) (gt, x, 0,  )
(2) (if_f, (1), (11) )
(3) (asn, 1,fact )
(4) (mul, fact, x)
(5) (asn, (4), fact )
(6) (sub, x, 1 )
(7) (asn, (6), x,  )
(8) (eq, x, 0 )
(9) (if_f, (8), (4))
(10) (wri, fact, _ )
(11) (halt, _, _)
```

Note: Triples have one major drawback， any movement of their positions becomes difficult.

8.1.3 P-Code

- P-code began as a standard target assembly code produced by a number of Pascal compilers of the 1970s and early 1980s.

- It was designed to be the actual code for a hypothetical stack machine, called the P-machine, for which an interpreter was written on various actual machines.

- The P-machine consists of a code memory, an unspecified data memory for named variables, and a stack for temporary data, together with whatever registers are needed to maintain the stack and support execution.

8.1.3 P-Code

*2*a+(b-3)*

- ldc 2; load constant 2 (pushes 2 onto the temporary)
- lod a; load value of variable a(pushes a onto the temporary)
- mpi; integer multiplication (pops these two values from the stack, multiplies them (in reverse order), and pushes the result onto the stack.)
- lod b ; load value of variable b
- ldc 3 ; load constant 3
- sbi ; integer subtraction(subtracts the first from the second)
- adi ; integer addition

8.1.3 P-Code

- x:= y+1
- lda x ;load address of x
- lod y ; load value of y
- ldc 1 ;load constant 1
- adi ;add
- sto ;store top to address below top & pop both

8.1.3 P-Code

- P-code is in many respects closer to actual machine code than three-address code. P-code instructions also require fewer addresses:
- P-code is less compact than three-address code in terms of numbers of instructions, and P-code is not “ self-contained ” in that the instructions operate implicitly on a stack.
- Historically, P-code has largely been generated as a text file, but the previous descriptions of internal data structure implementations for three-address code will also work with appropriate modification for P-code.

8.2 Basic Code Generation Techniques

8.2.1 Intermediate Code or Target Code as a Synthesized Attribute

- Intermediate code generation can be viewed as an attribute computation. This code becomes a synthesized attribute that can be defined using an attribute grammar and generated either directly during parsing or by a postorder traversal of the syntax tree.

8.2.1 Intermediate Code or Target Code as a Synthesized Attribute

exp → *id* = *exp* | *aexp*

aexp → *aexp* + *factor* | *factor*

factor → (*exp*) | *num* | *id*

- 1) **P-Code**

  The existence of embedded assignments is a complicating factor.

  - stn : stores the value to the address but leaves the value at the top of the stack, while discarding the address.
  - ++: instructions are to be concatenated with newlines inserted between them.
  - ||:when a single instruction is being built and a space is to be inserted.
- 2) **Three-Address Code**

  Requires the attribute grammar include a new name attribute for each node.

8.2.1 Intermediate Code or Target Code as a Synthesized Attribute

(x=x+3)+4

- lda x
- lod x
- ldc 3
- adi
- stn
- ldc 4
- adi

8.2.2 Practical Code Generation

The basic algorithm can be described as the following recursive procedure.

```
Procedure gencode (T: treenode);
        Begin
         If T is not nil then
                Generate code to prepare for code of left child of T;
                Gencode(left child of T);
                Generate code to prepare for code of right child of T;
                Gencode(right child of T);
                Generate code to implement the action of T;
        End;
```

8.2.2 Practical Code Generation

```
● typedef enum {Plus, Assign} optype;
● typedef enum {OpKind, ConstKind, IdKind} NodeKind;
● typedef struct streenode{
        NodeKind kind;
        Optype op;  /* used with OpKind */
        struct streenod *lchild, *rchild;
        int val; /* used with ConstKind */
        char * strval;
        /* used for identifiers and numbers */
  } STreenode;
● typedef STreenode *syntaxtree;
```

8.2.2 Practical Code Generation

- 1: the code uses the standard C function *sprintf* to concatenate strings into the local temporary *codestr.*
- 2: the procedure *emitcode* is called to generate a single line of P-code, either in a data structure or an output file.
- 3: the two operator cases (*plus* and *assign*) require two different traversal orders, plus needs only postorder, whiling assign requires some preorder and some postorder processing.

- %{
- #define YYSTYPE char *
- /* make Yacc use strings as values */
- /* other inclusion code … */
- %}
- %token NUM ID
- %%
- exp : ID
- { sprintf (codestr, "%s %s", "lda", $1);
- emitCode (codestr); }
- ' = ' exp
- { emitCode ("stn"); }
- | aexp
- ;
- /* */

```
aexp  : aexp  ‘+’  factor  { emitCode(“adi”);  }
          | factor
          ;
factor: ‘(’ exp  ‘)’
          | num   {  sprintf(codestr, “%s   %s”,   “ldc”, $1);
                              emitCode(codestr);   }
          | ID    {  sprintf(codestr, “%s   %s”,   “lod”, $1);
                              emitCode(codestr);   }
          ;
%%
```

8.2.3 Generation of Target Code from Intermediate Code

- The final code generation pass must supply all the actual locations of variables and temporaries, plus the code necessary to maintain the runtime environment.

8.2.3 Generation of Target Code from Intermediate Code

- Code generation from intermediate code involves either or both of two standard techniques: ***macro expansion*** and ***static simulation***.

8.2.3 Generation of Target Code from Intermediate Code

1. Macro expansion involves replacing each kind of intermediate code instruction with an equivalent sequence of target code instructions.

2. Static simulation involves a straight-line simulation of the effects of the intermediate code and generating target code to match these effects.

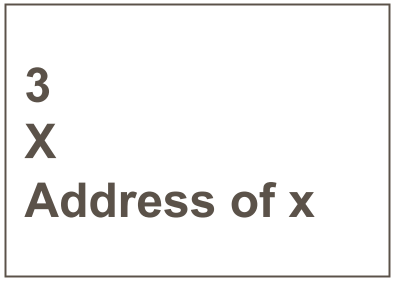
8.2.3 Generation of Target Code from Intermediate Code

```
lda x
lod x
ldc 3
adi
stn
ldc 4
adi
```

$\Longrightarrow$

```
t1 = x+3
x = t1
t2 =t1 + 4
```

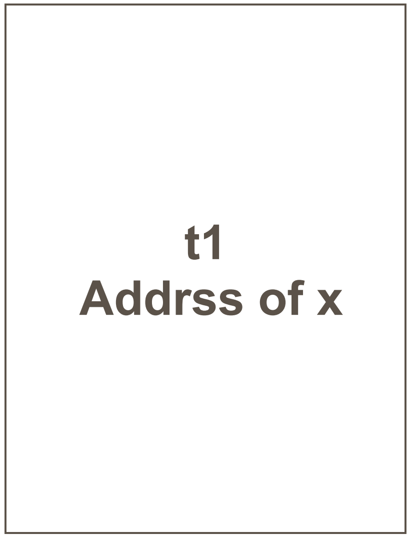
8.2.3 Generation of Target Code from Intermediate Code



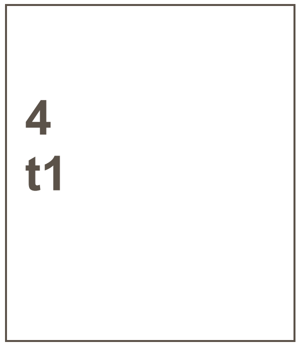
The adi operation is processed , the three-address instruction **t1=x+3** is generated .

8.2.3 Generation of Target Code from Intermediate Code

- The stn instruction then causes the three-address instruction   x=t1



8.2.3 Generation of Target Code from Intermediate Code



- Finally, the adi instruction causes the three address indtruction
- t2=t1+4

8.2.3 Generation of Target Code from Intermediate Code

```
a = b+c
```

```
lda a
lod b
lod c
adi
sto
```

8.3 Code Generation of Data Structure References

8.3.1 Address Calculations

1) Three-Address Code for Address Calculations

Suppose we wished to store the constant value 2 at the address of the variable x plus 10 bytes.

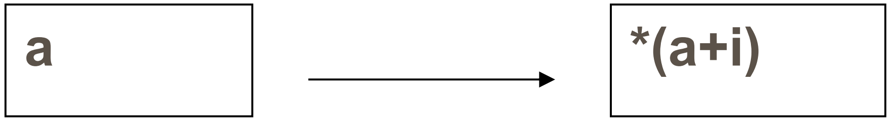
```
t1 = &x +10
*t1 = 2
```

The implementation of these new addressing modes requires that the data structure for three-address code contain a new field or fields. For example, the quadruple data structure of Figure 8.4 can be augmented by an enumerated *AddrMode* field with possible values *None*, *Address*, and *Indirect*.

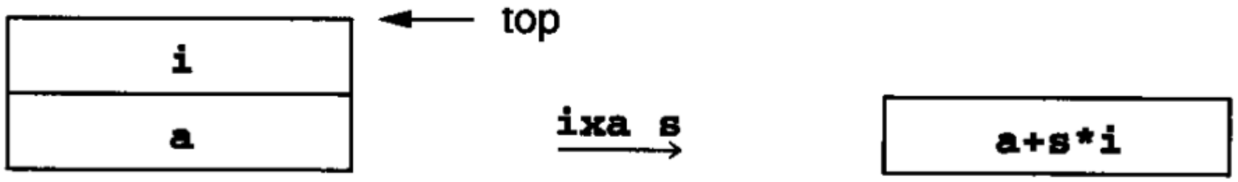
8.3.1 Address Calculations

2) P-Code for Address Calculations

1. ind (“ indirect load”) ind i



2. ixa (“indexed address”)



8.3.2 Array References

Each address must be computed from the base address of a (its starting address in memory) and an offset that depends linearly on the value of the subscript.

8.3.2 Array References

The offset is computed from the subscript value as follows.

1. An adjustment must be made to the subscript value if the subscript range does not begin at 0.
2. The adjusted subscript value must be multiplied by a scale factor that is equal to the size of each array element in memory.

Finally, the resulting scaled subscript is added to the base address to get the final address of the array element.

8.3.2 Array References

The address of an array element a[t]

base _ address (a) + (t - lower_bound (a)) * element_size (a)

ways of expressing this address calculation in three-address code and P-code.

&a in three-address code is the same as ***base-address(a)***

lda a loads the base address of a onto the P-machine stack.

element-size(a): the element size of array a on the target machine. Since this is a static quantity (assuming static typing), this expression will be replaced by a constant at compile time.

8.3.2 Array References

1) Three-Address Code for Array References

introduce two new operations, one that fetches the value of an array element

t2= a[t1]

and one that assigns to the address of an array element

a[t2]= t1

these could be given the symbols =[] and []=

a[i+1] = a [j*2]+3

translate into the three-address instructions

```
t1 = j * 2
t2 = a [t1]
t3 = t2 + 3
t4 = i + 1
a [t4] = t3
```

8.3.2 Array References

- We may also write out the address computations of an array element directly in three-address code.

t2 = a[t1]

```
    t3 = t1 * elem_size(a)
    t4 = &a +t3
    t2 = *t4
```

a[t2] = t1

```
    t3 = t2*elem_size(a)
    t4= &a + t3
    *t4 = t1
```

8.3.2 Array References

a[i+1] = a[j+2] + 3

```
t1 = j * 2
t2 = t1 * elem_size(a)
t3 = &a + t2
t4 = *t3
t5 = t4 + 3
t6 = i + 1
t7 = t6 * elem_size (a)
t8 = &a + t7
*t8 = t5
```

8.3.2 Array References

a[t2] = t1 is written in P-code as

```
lda a
lod t2
ixa elem-size(a)
lod t1
sto
```

8.3.2 Array References

*a[i+1] = a[j*2] + 3* translates to the following P-code instructions:

```
lda a
lod i
ldc 1
adi
ixa elem_size(a)
lda a
lod j
ldc 2
mpi
ixa elem_size(a)
ind 0
ldc 3
adi
sto
```

8.3.2 Array References

- *3)* A Code Generation Procedure with Array References

$exp \rightarrow subs = exp \mid aexp$

$aexp \rightarrow aexp + factor \mid factor$

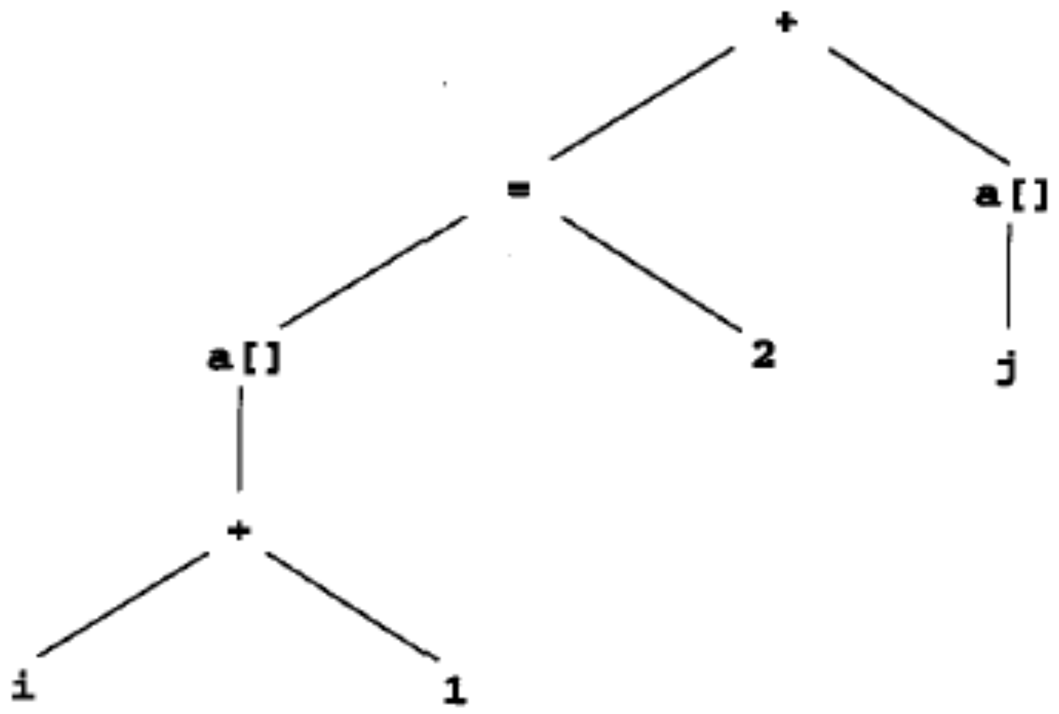
$factor \rightarrow (exp) \mid num \mid subs$

$subs \rightarrow id \mid id[exp]$

```
typedef enum { Plus, Assign, Subs } Optype;
```

8.3.2 Array References

- (a [i + 1] = 2) + a [j]



8.3.2 Array References

The following P-code for *(a [i + 1] = 2) + a [j]* **:**

```
lda a
lod i
ldc 1
adi
ixa elem_size(a)
ldc 2
stn
lda a
lod j
ixa elem_size(a)
ind 0
adi
```

Function gencode( ) -- figure 8.9

8.4 Code Generation of Control Statements and Logical Expressions

- if-statement and while-statement.
- Intermediate code generation for control statements involves the generation of labels in manner.

8.4.1 Code Generation for If – and While – Statements

if-stmt → **if** (*exp*) *stmt* | **if** (*exp*) *stmt* **else** *stmt*

while-stmt → **while** (*exp*) *stmt*

- The chief problem is to translate the structured control features into an "unstructured" equivalent involving jumps, which can be directly implemented.
- Compilers arrange to generate code for such statements in a standard order that allows the efficient use of a subset of the possible jumps that a target architecture might permit.

8.4.1 Code Generation for If – and While – Statements

In each of these arrangements, there are only two kinds of jumps---

- unconditional jumps
- jumps when the condition is false----the true case is always a “fall-through” case that needs no jump.

This reduces the number of the jumps that the compiler needs to generate. It only needs two jump instructions.

- False jumps: *if_f, t1,L1,_* (three address)

  fjp L1 (P-code)
- unconditional jumps: *goto* (three address)

  ujp (P-code)

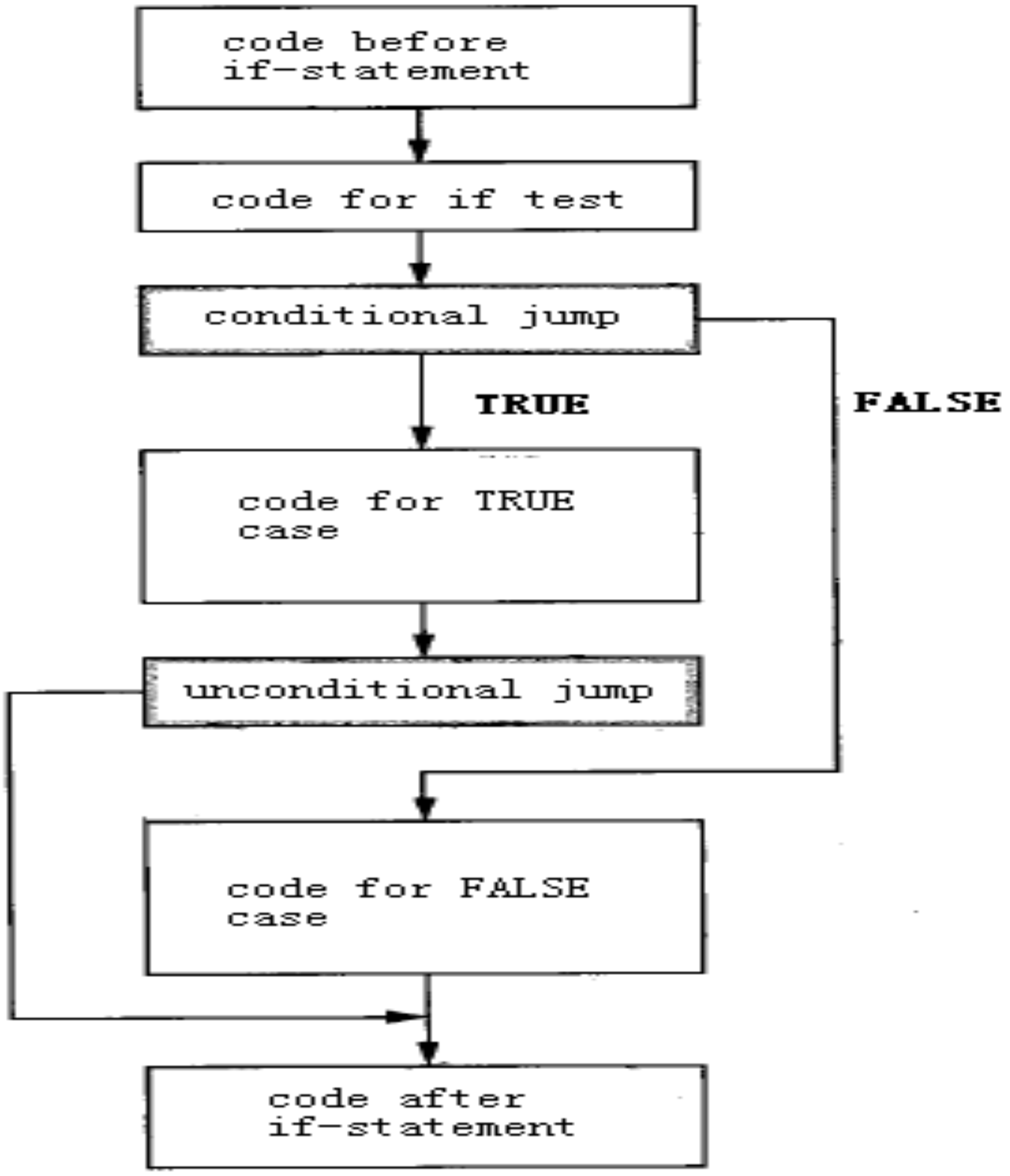


Figure 8.10 Typical code arrangement for an if-statement

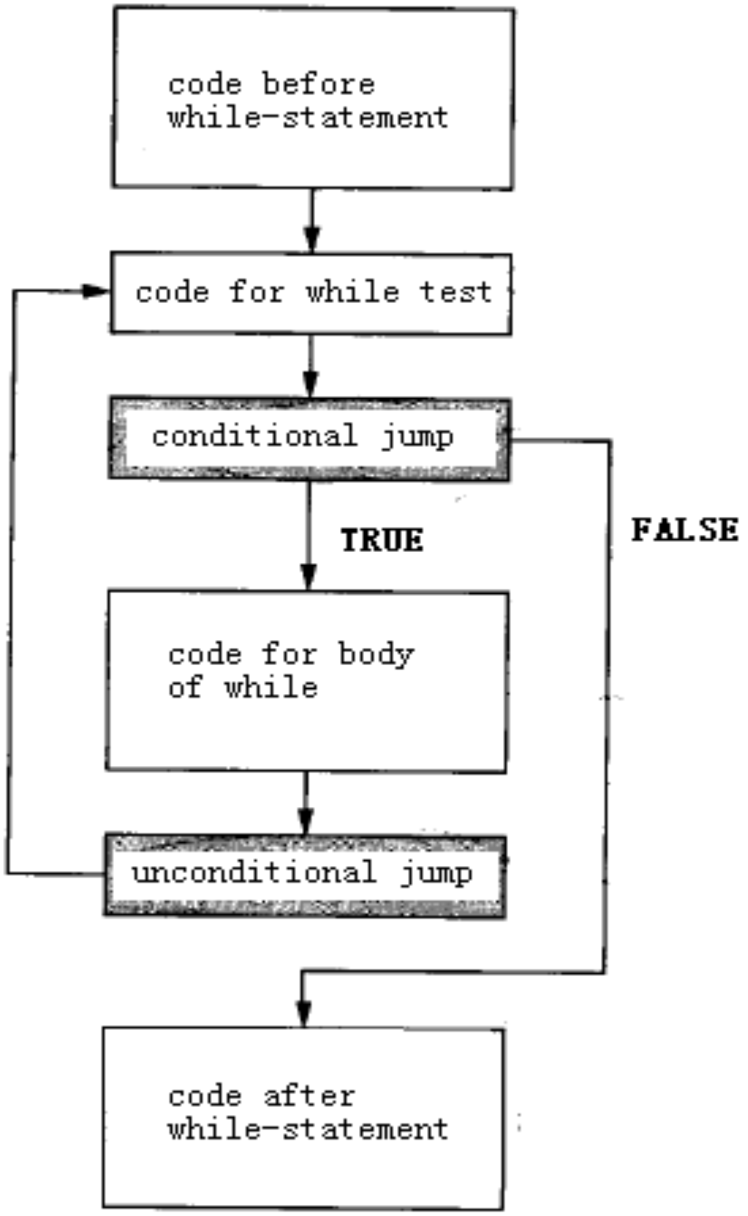


Figure 8.11 Typical code arrangement for a while-statement

Three-Address Code for Control Statement

if (E) S1 else S2

the following code pattern is generated:

```
<code to evaluate E to t1>
if_false t1 goto L1
<code for S1>
goto L2
label L1
<code for S2>
label L2
```

Three-Address Code for Control Statement

while (E) S

the following three-address code pattern to be generated:

```
label L1
<code to evaluate E to t1>
if_false t1 goto L2
<code for S>
goto L1
label L2
```

P-Code for Control Statement

For the statement

if (E) S1 else S 2

The following P-code pattern is generated:

```
<code to evaluate E>
fjp L1
<code for S 1>
ujp L2
lab L1
<code for S 2>
lab L2
```

P-Code for Control Statement

while (E) S

the following P-code pattern is generated:

```
lab L1
<code to evaluate E>
fjp L2
<code for S>
ujp L1
lab L2
```

8.4.2 Generation of Labels and Backpatching

- One feature of code generation for control statements : jumps to a label must be generated prior to the definition of the label itself.
- A standard method for generating such forward jumps is either to leave a gap in the code where the jump is to occur or to generate a dummy jump instruction to a fake location. Then, when the actual jump location becomes known, this location is used to fix up, or backpatch, the missing code.

8.4.2 Generation of Labels and Backpatching

- During the backpatching process a further problem may arise in that many architectures have two varieties of jumps, a short jump or branch (within 128 bytes of code) and a long jump that requires more code space . In that case, a code generator may need to insert nop instructions when shortening jumps, or make several passes to condense the code.

8.4.3 Code Generation of Logical Expressions

- The standard way to do this is to represent the Boolean value false as 0 and true as 1. Then standard bitwise *and* and *or* operators can be used to compute the value of a Boolean expression on most architectures.
- A further use of jumps is necessary if the logical operations are short circuit. For example, if *a* is a Boolean expression that is computed to be false, then the Boolean expression *a and b* can be immediately determined to be false without evaluating *b*.

8.4.3 Code Generation of Logical Expressions

Short-circuit Boolean operators are similar to if-statements, except that they return values, and often they are defined using if-expressions as

a and b ≡ if a then b else false

a or b ≡ if a then true else b

To generate code that ensures that the second subexpression will be evaluated only when necessary, we must use jumps in exactly the same way as in the code for if-statements.

8.4.3 Code Generation of Logical Expressions

- (x ! = 0) & & (y = = x) is:

```
lod x
ldc 0
neq
fjp L1
lod y
lod x
equ
ujp L2
lab L1
lod FALSE
lab L2
```

8.4.4 A Sample code Generation Procedure for If- and While- Statements

- *stmt* → *if-stmt* | *while-stmt* | **break** | **other**
- *if-stmt* → **if** (*exp*) *stmt* | **if** (*exp*) *stmt* **else** *stmt*
- *while-stmt* → **while** (*exp*) *stmt*
- *exp* → **true** | **false**

8.4.4 A Sample code Generation Procedure for If- and While- Statements

if (true) while (true) if (false) break else other

it generates the code sequence

```
ldc true      (if……..
fjp L1
lab L2       (while……..
ldc true
fjp L3
ldc false     (if………
fjp L4
ujp L3
ujp L5
lab L4
Other
lab L5
ujp L2
lab L3
lab L1
```

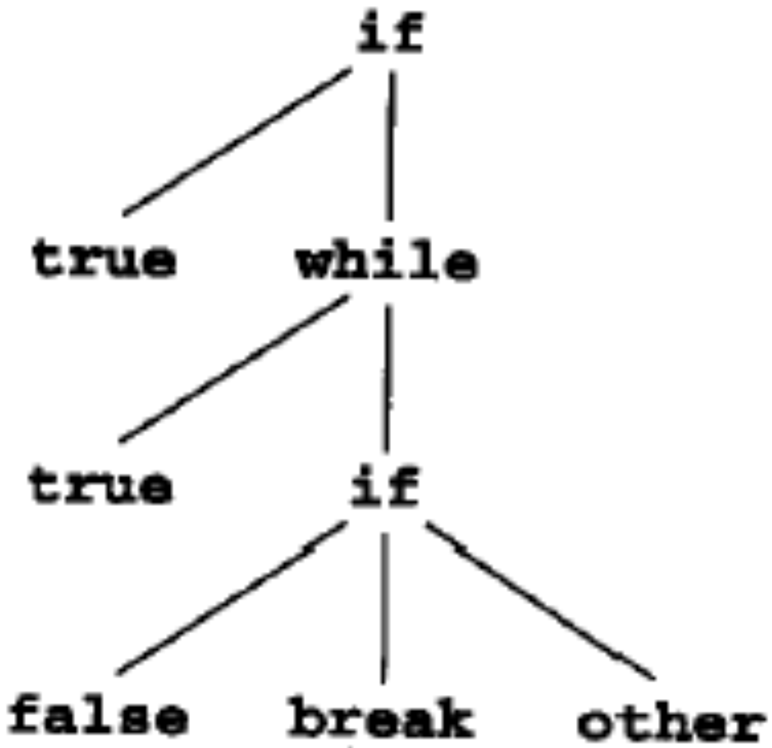
8.4.4 A Sample code Generation Procedure for If- and While- Statements

The following C declaration can be used to implement an abstract syntax tree:

```
typedef enum { ExpKind,IfKind,
      WhileKind, BreakKind, OtherKind } NodeKind;
typedef struct streenode
            { NodeKind kind;
              struct streenode * child[3] ;
              int val; /* used with ExpKind */
            } STreeNode;
typedef STreeNode * SyntaxTree;
```

8.4.4 A Sample code Generation Procedure for If- and While- Statements

- if (true) while (true) if (false) break else other



8.4.4 A Sample code Generation Procedure for If- and While- Statements

- The **genlabel** procedure that returns **label** names in the sequence L1,L2,L3, and so on.
- The **gencode** procedure here has an extra **label** parameter that is needed to generate an absolute jump for a break-statement.
- A break-statement will always cause a jump out of the most closely nested while-statement.