Chapter 8
Code Generation

한양대학교 컴퓨터공학부
컴파일러
2014년 2학기

Overview

- Intermediate code
  - three-address code and P-code
- Basic code generation
- Code generation of data structure references
- Code generation of control statements and logical expressions

syntax tree -> intermediate code -> target code

Intermediate code resembles target code

- New form of intermediate representation from the syntax tree
  not resembles target code
- Some form of linearization of the syntax tree
- Information contained in the symbol table
  - Scopes
  - Nesting levels
  - Offsets of variables
- Making a compiler more easily retargetable

  intermediate code -> target code가 entire code -> target code 보다 쉽기 때문에

Three-address code

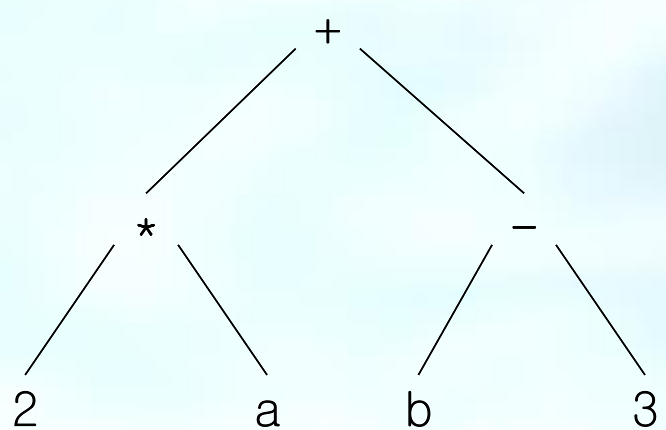
- General form

  x = y *op* z

- Expression

  2*a+(b-3)

- Syntax tree



Three-address code


```
t1 = 2 * a
t2 = b - 3
t3 = t1 + t2
```

```
t1 = b – 3
t2 = 2 * a
t3 = t2 + t1
```

Three-address code


```
{   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
```

```
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
```

Quadruple implementation

- quadruple: one for the operation and three for the addresses

read x	(rd, x, _, _)
t1 = x > 0	(gt, x, 0, t1)
if_false t1 goto L1	(if_f, t1, L1, _)
fact = 1	(asn, 1, fact, _)
label L2	(lab, L2, _, _)
t2 = fact * x	(mul, fact, x, t2)
fact = t2	(asn, t2, fact, _)
t3 = x -1	(sub, x, 1, t3)
x = t3	(asn, t3, x, _)
t4 = x == 0	(eq, x, 0, t4)
if_false t4 goto L2	(if_f, t4, L2, _)
write fact	(wri, fact, _, _)
label L1	(lab, L1, _, _)
halt	(halt, _, _, _)

C code data structures


```
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;
```

Triple implementation

to use the instructions themselves to represent the temporaries

```
(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, _, _, _)
```

```
(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, _, _)
```

temporary & local 제거됨

P-Code

- a standard target assembly code in 1970s and 1980s
- designed to be the actual code for a hypothetical stack machine, called the **P-machine**
- portable

P-Code

- Expression

2*a+(b-3)

x := y +1

- P-code

ldc 2	; load constant 2	lda x	; load address of x
lod a	; load value of variable a	lod y	; load value of y
mpi	; integer multiplication	ldc 1	; load constant 1
lod b	; load value of variable b	adi	; add
ldc 3	; load constant 3	sto	; store top to address
sbi	; integer subtraction		; below top & pop both
adi	; integer addition		

Figure 8.6


```
lda x        ; load address of x
rdi          ; read an integer, store to
             ; address on top of stack (& pop it)
------------------------------------------------------
lod x        ; load the value of x
ldc 0        ; load constant 0
grt          ; pop and compare top two values
             ; push Boolean result
fjp L1       ; pop Boolean value, jump to L1 if false
------------------------------------------------------
lda fact     ; load address of fact
ldc 1        ; load constant 1
sto          ; pop two values, storing first to
             ; address represented by second
------------------------------------------------------
lab L2       ; definition of label L2
------------------------------------------------------
lda fact     ; load address of fact
lod fact     ; load value of fact
lod x        ; load value of x
mpi          ; multiply
sto          ; store top to address of second & pop
------------------------------------------------------
lda x        ; load address of x
lod x        ; load value of x
ldc 1        ; load constant 1
sbi          ; subtract
sto          ; store (as before)
------------------------------------------------------
lod x        ; load value of x
ldc 0        ; load constant 0
equ          ; test for equality
fjp L2       ; jump to L2 if false
------------------------------------------------------
lod fact     ; load value of fact
wri          ; write top of stack & pop
lab L1       ; definition of Label L1
stp
```

```
read x;
if 0 <x then
    fact := 1;
    repeat
        fact := fact * x;
        x := x -1
    until x = 0;
    write fact
end
```

Basic code generation techniques

- **Grammar**

$$exp \rightarrow \textbf{id} = exp \mid aexp$$

$$aexp \rightarrow aexp + factor \mid factor$$

$$factor \rightarrow \textbf{(}\ exp\ \textbf{)} \mid \textbf{num} \mid \textbf{id}$$

intermediate code generation(or direct target code generation without intermediate code) can be viewed as an attribute computation similar to many of the attribute problem

if the generated code is viewed as a string attribute (with instructions separate by newline characters) , then 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

viewing code generation as the computation of a synthesized string attribute
장점 : 1. show clearly the relationship among the code sequences of different parts of the syntax tree and for comparing different code generation methods.

단점 : 1. an inordinate amount of string copying and wasted memory
2. code generation in general depends heavily on inherited attributes and this greatly complicates the attribute grammars.

Basic code generation techniques

- sto → destructive stack에 아무것도 남지 않는다.
- stn → nondestructive store store the value to the address but leaves the value at the top of the stack, while discarding the address
- Expression

  (x=x+3)+4

- P-code attribute

  lda x
  lod x
  ldc 3
  adi
  stn
  ldc 4
  adi

<destructive>
lda x
lod x
ldc 3
adi
sto
lod x
ldc 4
adi

Synthesized attribute for P-code

pcode - attribute name for the p-code string
++ - instructions are to be concatenated with newlines
|| - a single instruction is being build and a space is to be inserted

Grammar Rule	Semantic Rules
$exp_1 \rightarrow$ ***id*** = exp_2	$exp_1.pcode$ = "**lda**" || ***id***.*strval* ++ $exp_2.pcode$ ++ "**stn**"
exp $\rightarrow$ *aexp*	*exp.pcode* = *aexp.pcode*
$aexp_1 \rightarrow aexp_2$ + *factor*	$aexp_1.pcode = aexp_2.pcode$ ++ *factor.pcode* ++ "**adi**"
aexp $\rightarrow$ *factor*	*aexp.pcode* = *factor.pcode*
factor $\rightarrow$ **(** *exp* **)**	*factor.pcode* = *exp.pcode*
factor $\rightarrow$ ***num***	*factor.pcode* = "**ldc**" || ***num***.*strval*
factor $\rightarrow$ ***id***	*factor.pcode* = "**lod**" || ***id***.*strval*

Attribute grammar for three-address code

Grammar Rule	Semantic Rules
$exp_1 \rightarrow \mathbf{id} = exp_2$	$exp_1.name = exp_2.name$ $exp_1.tacode = exp_2.tacode$ ++ $\mathbf{id}.strval$ ||"="||$exp_2.name$
$exp \rightarrow aexp$	$exp.name = aexp.name$ $exp.tacode = aexp.tacode$
$aexp_1 \rightarrow aexp_2 + factor$	$aexp_1.name = newtemp()$ $aexp_1.tacode =$ $aexp_2.tacode$ ++ $factor.tacode$ ++ $aexp_1.name$ ||"="||$aexp_2.name$ ||"+"||$factor.name$
$aexp \rightarrow factor$	$aexp.name = factor.name$ $aexp.tacode = factor.tacode$
$factor \rightarrow (\ exp\ )$	$factor.name = exp.name$ $factor.tacode = exp.tacode$
$factor \rightarrow \mathbf{num}$	$factor.name = \mathbf{num}.strval$ $factor.tacode$ = " "
$factor \rightarrow \mathbf{id}$	$factor.name = \mathbf{id}.strval$ $factor.tacode$ = " "

Practical code generation

- modifications of the postorder traversals
- 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;
```

definitions for an abstract syntax tree

- C definitions

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

- Expression
  (x=x+3)+4
- Syntax tree

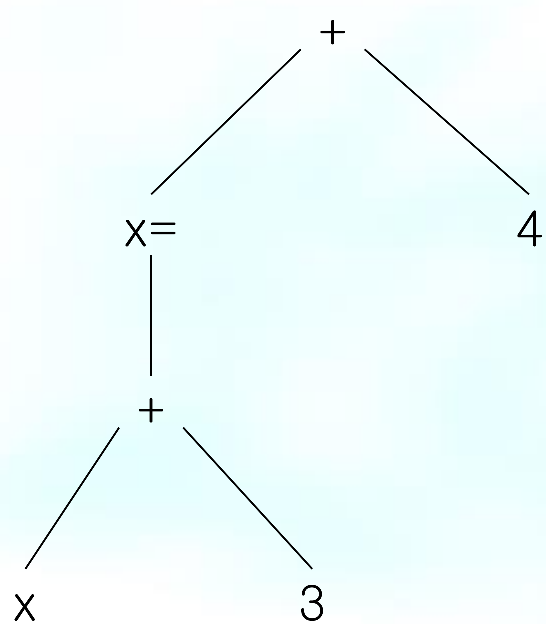


Figure 8.7


```
void genCode(  SyntaxTree t)
{  char codestr[CODESIZE];
     /* CODESIZE = max length of 1 line of P-code */
     if  (t !=NULL)
     {  switch  (t->kind)
          {  case OpKind:
               switch (t->op)
               {  case Plus:
                    genCode (t->lchild);
                    genCode (t->rchild);
                    emitCode ("adi");
                    break;
                  case Assign:
                    sprintf (codestr, "%s %s", "lda", t->strval);
                    emitCode (codestr);
                    genCode (t->lchild);
                    emitCode ("stn");
                    break;
                  default:
                    emitCode ("Error");
                    break;
               }
               break;
            case ConstKind:
               sprintf (codestr, "%s %s", "ldc", t->strval);
               emitCode (codestr);
               break;
            case IdKind:
               sprintf (codestr, "%s %s", "lod", t->strval);
               emitCode (codestr);
               break;
            default:
               emitCode ("Error");
               break;
          }
     }
}
```

Figure 8.8


```
%{
#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);  }
                ;
%%
/*  utility function . . .  */
```

Generation of Target Code from Intermediate Code

- **Expression**

  (x=x+3)+4

- **P-code**

  lda x

  lod x

  ldc 3

  adi

  stn

  ldc 4

  adi

- Three-address code

  t1 = x + 3

  x = t1

  t2 = t1 + 4

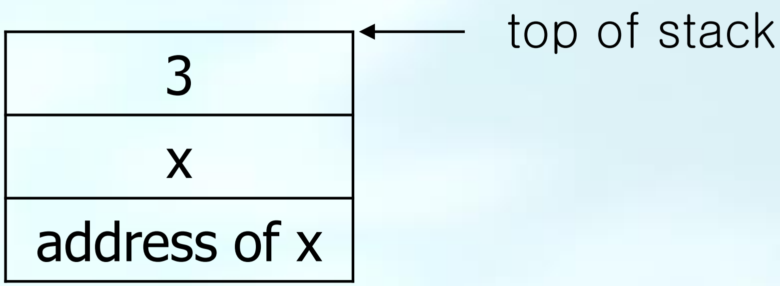
- **P-code**

  lda x
  lod x
  ldc 3
  adi
  stn
  ldc 4
  adi

- Three-address code

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

- P-machine stack



- **P-code**

  lda x
  lod x
  ldc 3
  adi
  stn
  ldc 4
  adi

- Three-address code

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

- P-machine stack

t1	← top of stack
address of x	

- **P-code**
  - lda x
  - lod x
  - ldc 3
  - adi
  - stn
  - ldc 4
  - adi

- Three-address code
  - t1 = x + 3
  - x = t1
  - t2 = t1 + 4

- P-machine stack

t1 ← top of stack

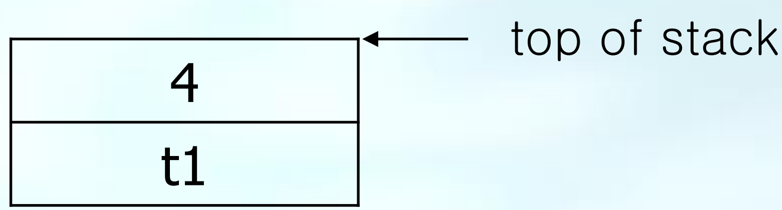
- **P-code**

  lda x
  lod x
  ldc 3
  adi
  stn
  ldc 4
  adi

- Three-address code

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

- P-machine stack



- **P-code**

  lda x
  lod x
  ldc 3
  adi
  stn
  ldc 4
  adi

- Three-address code

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

- P-machine stack

t2	← Top of stack

- **Three-address code**

  a = b + c

- **P-code**

```
lda a
lod b ; or ldc b if b is a const
lod c ; or ldc c if c is a const
adi
sto
```

P-code

```
lda t1
lod x
ldc 3
adi
sto
---------
lda x
lod t1
sto
---------
lda t2
lod t1
ldc 4
adi
sto
```

Three-address code

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

- **P-code**

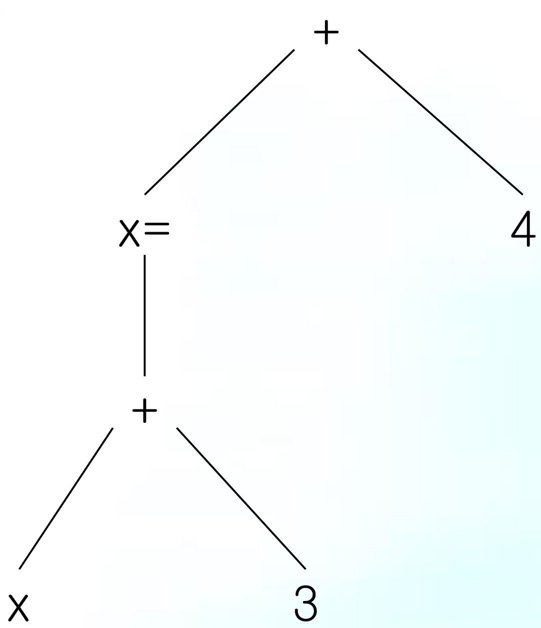
```
lda t1
lod x
ldc 3
adi
sto
```

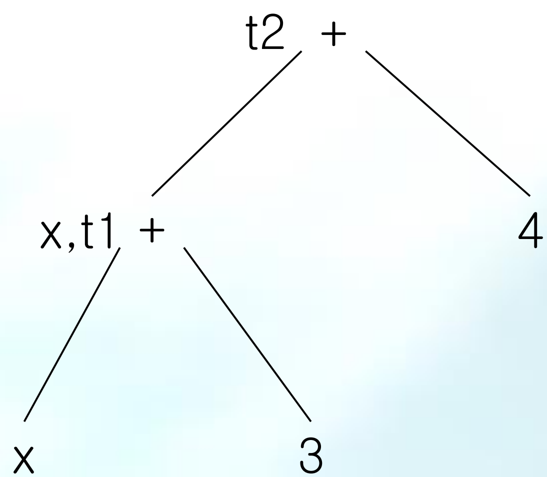
```
lda x
lod t1
sto
```

```
lda t2
lod t1
ldc 4
adi
sto
```

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

- Resulting tree





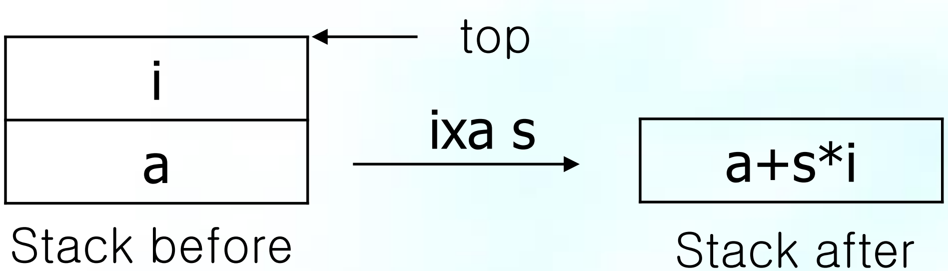
Code generation of data structure references

- ind (index) - index 값 처리할때 사용

ex) 1. x = a[3]
lda x
lda a
ind 3
sto

ex) index -> 값
lda x
ind 0 = lod x

- ixa i번째 index의 주소를 알때 사용



- **Three-address code**

  t1 = &x + 10

  *t1 = 2

- **P-code**

  lda x

  ldc 10

  ixa 1

  ldc 2

  sto

```
t1 = &x + 10
lda t1
lda x
ldc 10
adi
sto
```

```
*t1 = 2
lod t1
ldc 2
sto
```

Array references

- C code

```
int a[SIZE]; int i, j;
. . .
a[i+1] = a[j*2] + 3;
```

- address of a[i+1]

  addr(a[0]) + (i + 1) * sizeof(int)

- address of a[t]

  base_address(a)+(*t*–*lower_bound*(a))**element_size*(a)

- **Source code**

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

- **Three-address instructions**

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

- **Assignment**

```
t2 = a[t1]
```

- **Written**

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

- **Assignment**

  a[t2] = t1

- **Written**

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

- **Source code**

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

- **Three-address instructions**

```
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
```

a[j*2]

addr of a[i+1]

- **Array reference**

  t2 = a[t1]

- **P-code**

```
lda t2
lda a
lod t1
ixa elem_size(a)
ind 0
sto
```

- **Array assignment**

  a[t2] = t1

- **P-code**

```
lda a
lod t2
ixa elem_size(a)
lod t1
sto
```

- **Source code**

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

- **P-code**

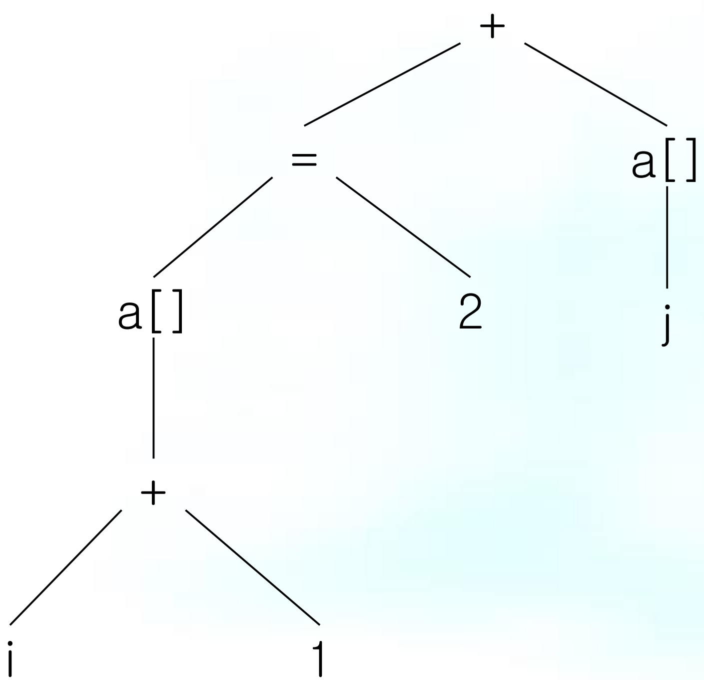
```
lda a
lod i
ldc 1
adi
ixa elem_size(a)
lda a
lod j
ldc 2
mpi
ixa elem_size(a)
ind 0
idc 3
adi
sto
```

Code Generation with array

- **Grammar**

$$
\begin{aligned}
&exp \to subs \;\mathbf{=}\; exp \mid aexp \\
&aexp \to aexp \;\mathbf{+}\; factor \mid factor \\
&factor \to \mathbf{(}\; exp \;\mathbf{)} \mid \boldsymbol{num} \mid subs \\
&subs \to \boldsymbol{id} \mid \boldsymbol{id}\,\mathbf{[}\; exp \;\mathbf{]}
\end{aligned}
$$

- **Expression**
  (a[i+1] = 2) + a[j]
- **Syntax tree**



- P-code

```
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
```

Figure 8.9


```
void genCode(  SyntaxTree t, int isAddr)
{  char codestr[CODESIZE];
     /* CODESIZE = max length of 1 line of p-code */
     if  (t !=NULL)
     {  switch  (t->kind)
          {  case OpKind:
               switch (t->op)
               {  case Plus:
                      if (isAddr) emitCode("Error");
                      else { genCode(t->lchild, FALSE);
                               genCode(t->rchild, FALSE);
                               emitCode ("adi");}
                      break;
                  case Assign:
                      genCode (t->lchild, TRUE);
                      genCode (t->rchild, FALSE);
                      emitCode ("stn");
                      break;
                  case Subs:
                      sprintf(codestr, "%s %s", "lda", t->strval);
                      emitCode(codestr);
                      genCode(t->lchild, FALSE);
                      sprintf(codestr, "%s%s%s", "ixa elem_size(",t->strval,")");
                      emitCode(codestr);
                      if (!isAddr) emitCode("ind 0");
```

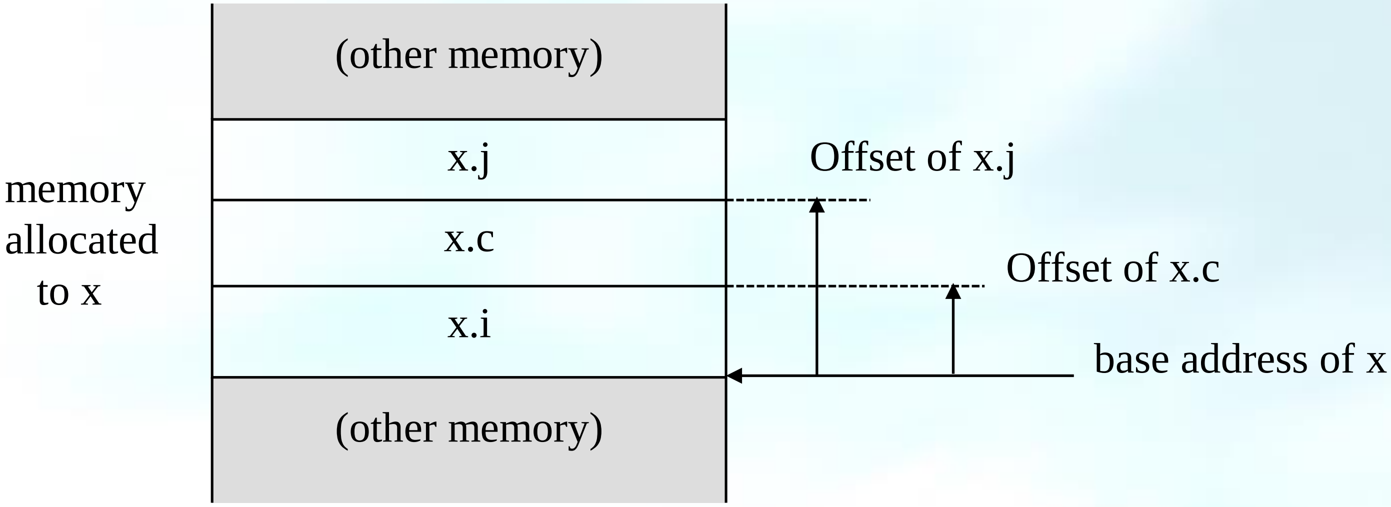
Figure 8.9


```
                break;
              default:
                emitCode ("Error");
                break;
            }
            break;
          case ConstKind:
               if (isAddr) emitCode("Error");
               else
               {  sprintf(codestr, "%s %s", "ldc", t->strval);
                  emitCode(codestr);
                }
                break;
          case IdKind:
                if (isAddr)
                   sprintf(codestr, "%s %s", "lda", t->strval);
                else
                   sprintf(codestr, "%s %s", "lod", t->strval);
                emitCode (codestr);
           break;
         default:
            emitCode ("Error");
            break;
       }
     }
  }
```

Record structure and pointer references

- **C declaration**

```
typedef struct rec
    {  int i;
       char c;
       int j;
     } Rec;
. . .
Rec x;
```



- **Three-address code**

  t1 = &x + field_offset(x, j)

- **Field assignment**

  x.j = x.i;

- **Three-address code**

  t1 = &x + field_offset(x,j)

  t2 = &x + field_offset(x,i)

  *t1 = *t2

Declaration in C

```
typedef struct treeNode
     {  int val;
        struct treeNode * lchild, * rchild;
      }  TreeNode;
. . .
TreeNode *p;
```

- **Assignments**

```
p->lchild = p;
p = p->rchild;
```

- **Three-address code**

```
t1 = p + field_offset(*p, lchild)
*t1 = p
t2 = p + field_offset(*p, rchild)
p = *t2
```

- **P-code**

  lda x

  ldc field_offset(x,j)

  ixa 1

- **Assignment**

  x.j = x.i;

- **P-code**

  lda x

  ldc field_offset(x,j)

  ixa 1

  lda x

  ind field_offset(x,i)

  sto

- **Assignment**

```
*x = i;
```

- **P code**

```
lod x
lod i
sto
```

- **Assignment**

```
i = *x;
```

- **P code**

```
lda i
lod x
ind 0
sto
```

- **Assignments**

```
p->lchild = p;
p = p->rchild;
```

- **P code**

```
lod p
ldc field_offset(*p, lchild)
ixa 1
lod p
sto
lda p
lod p
ind field_offset(*p, rchild)
sto
```

Code Generation of Control Statements and Logical Expression

- Jump Table
- Labels
- Backpatching
- Shortcut evaluation

Code Generation for If- and While-Statements

- **C-like syntax**

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

  while-stmt → while (*exp*) *stmt*

Code arrangement

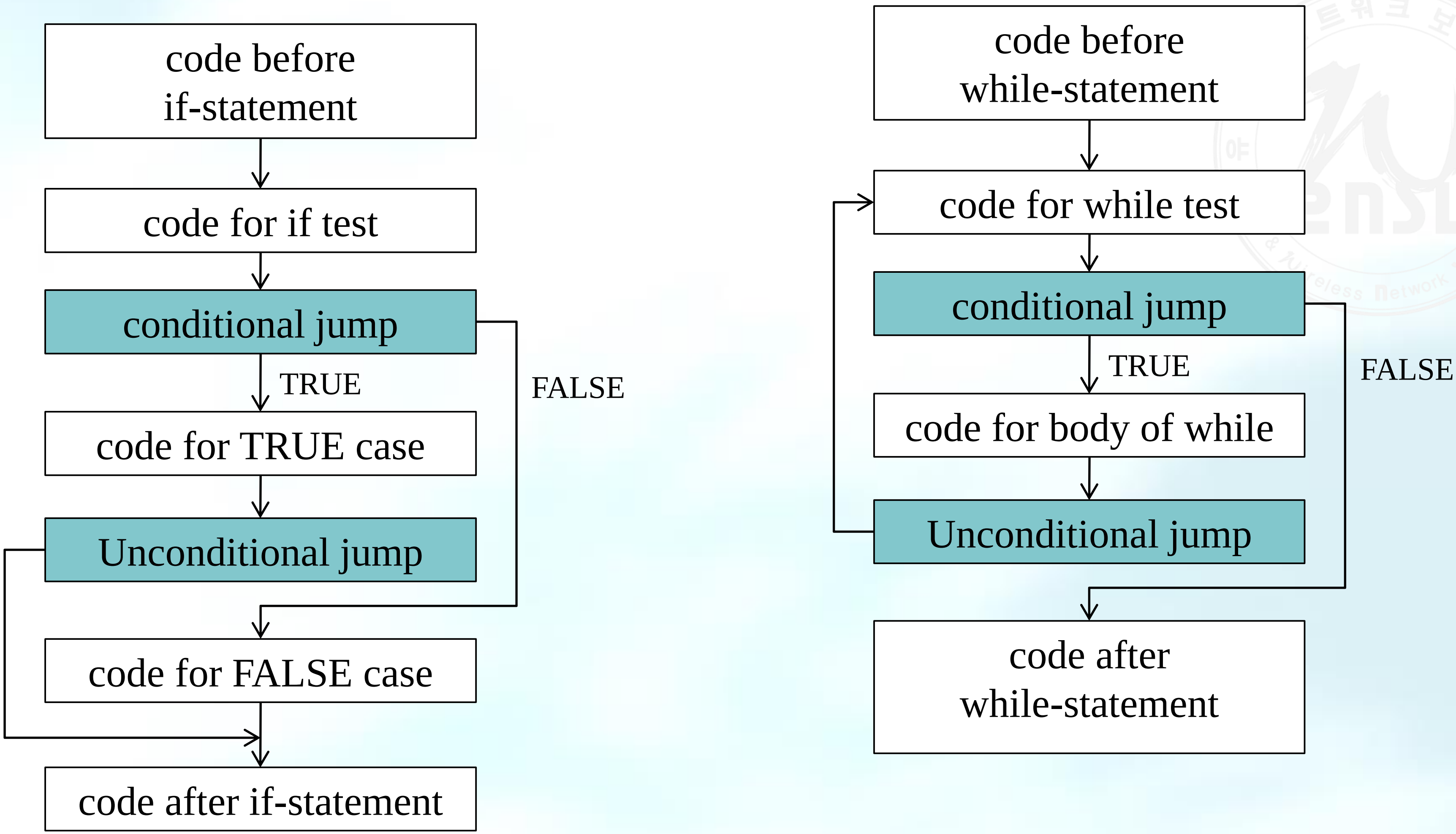


Fig 8.10 if-statement

Fig 8.11 while-statement

Three-Address Code for Control Statements

- **Statement**

  if (*E*) *S1* else *S2*

- **Three-Address Code**

  <code to evaluate *E* to t1>

  if_false t1 goto L1

  <code for *S1*>

  goto L2

  label L1

  <code for *S2*>

  label L2

While Statement – three-address code

- **Statement**

  while (*E*) *S*

- **Three-Address Code**

  label L1

  <code to evaluate *E* to t1>

  if_false t1 goto L2

  <code for *S*>

  goto L1

  label L2

If statement – P-code

- **Statement**

  if (*E*) *S1* else *S2*

- **P-Code**

  <code to evaluate *E*>

  fjp L1

  <code for *S1*>

  ujp L2

  lab L1

  <code for *S2*>

  lab L2

While Statement – P-code

- **Statement**

  while (*E*) *S*

- **P-Code**

  label L1

  <code to evaluate *E*>

  fjp L2

  <code for *S*>

  ujp L1

  lab L2

Code Generation of Labels and Backpatching

- **Target code generation**
  - Problem
    - Jumps to a label must be generated **prior to** the definition of the label
  - During target code generation
    - label can be simply passed on to an assembler if assembly code is generated,
  - During executable code generation
    - labels must be resolved into **absolute** or **relative** code locations

- **Method for generating forward jumps**
  - Leave a gap in the code where the jump is to occur
  - Generate a dummy jump instruction to a fake location
  - when the actual jump location becomes know, this location is used to fixed up the missing code → **backpatch**
  - **nop** instructions

Code Generation of Labels and Backpatching

- Backpatching process
  - Problem
    - Many architectures have two varieties of jumps
  - Short jump or branch
    - e.g. within 128 bytes of code
  - Long jump
    - requires more code
    - nop

Code Generation of Logical Expressions

- **Short circuit**
  - A logical operation is short circuit if it may fail to evaluate its second argument.

    if ((p!=NULL) && (p->val ==0))…

    Where evaluation of p->val when p is null cause memory fault.

- **Short circuit Boolean operator**
  - Similar to if-statements, except they return values

    a and $b \equiv$ if a then b else false

    a or $b \equiv$ if a then true else b

Example

- **Expression**

  (x!=0) && (y==x)

- **P-Code**

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

바이너리 코드 난독화

- 데드코드 인젝션 (Deadcode Injection)
- 컨트롤 플로우 난독화 (Control Flow Obfuscation)
- 레지스터 재할당 (Register Reassignment)
- 동일한 의미의 다른 수식 (Semantic Obfuscation)

데드코드 인젝션 (Deadcode Injection)

- 실제로는 실행되지 않는 명령어 또는 실행되더라도 아무런 의미가 없는 명령어를 삽입하여 난독화하는 방법

```
; Encode a single byte
ZipEncode proc a: BYTE
        mov     ecx, keys2
        and     ecx, 0ffffh

        or      ecx, 2

        mov     eax, ecx
        xor     ecx, 1
        xor     edx, edx
        mul     ecx

        shr     eax, 8
        push    eax


        invoke
ZipUpdateKeys, a
        pop     eax

        xor     al, a
        ret
ZipEncode endp
```

[Original Binary Code]

```
; Encode a single byte
ZipEncode proc a: BYTE
        mov     ecx, keys2
        and     ecx, 0ffffh
        nop
        or      ecx, 2
        inc     eax
        mov     eax, ecx
        xor     ecx, 1
        xor     edx, edx
        mul     ecx
        push    ecx
        shr     eax, 8
        push    eax
        inc     ecx
        dec     ecx
        invoke
ZipUpdateKeys, a
        pop     eax
        pop     ecx
        xor     al, a
        ret
ZipEncode endp
```

[Obfuscated Binary Code]

No Operation (NOP)

EAX의 값을 증가시켰으나 (INC EAX)
사용하지 않고 곧바로 ECX 값으로 덮어씀
(MOV EAX ECX)

ECX 값을 1 증가시켰으나 (INC ECX)
곧바로 1 감소시킴 (DEC ECX)

ECX값을 PUSH하였으나 (PUSH ECX)
사용하지 않고 POP하였음 (POP ECX)

컨트롤 플로우 난독화 (Control Flow Obfuscation)

- 코드의 여러 영역을 이동하며 실행되도록 난독화하는 방법
- C-Minus에서는 Unconditional Jump로 구현할 수 있음

```
; Encode a single byte
ZipEncode proc a: BYTE
        mov     ecx, keys2
        and     ecx, 0ffffh
        or      ecx, 2

        mov     eax, ecx
        xor     ecx, 1

        xor     edx, edx
        mul     ecx

        shr     eax, 8
        push    eax

        invoke
ZipUpdateKeys, a
        pop     eax
        xor     al, a

        ret
ZipEncode endp
```

[Original Binary Code]

```
; Encode a single byte
ZipEncode proc a: BYTE
        mov     ecx, keys2
        and     ecx, 0ffffh
        or      ecx, 2
        jmp     L1
L3:     shr     eax, 8
        push    eax
        jmp     L4
L1:     mov     eax, ecx
        xor     ecx, 1
        jmp     L2
L4:                     invoke
ZipUpdateKeys, a
        pop     eax
        xor     al, a
        jmp     L5
L2:     xor     edx, edx
        mul     ecx
        jmp     L3
L5:     ret
ZipEncode endp
```

[Obfuscated Binary Code]

실제로는 Jump 없이 실행될 수 있는 코드
난독화를 위해 코드를 여러 부분으로 자른
순서를 섞어, 이동하며 실행되도록 하였다.

레지스터 재할당 (Register Rearrangement)

- 레지스터의 용도를 변경하여 분석을 어렵게 하는 난독화 방법
- 예를 들어, Stack Base Register를 General Purpose Register로 간주하여 사용하는 방법 등이 있음

```
; Encode a single byte
ZipEncode proc a: BYTE

        mov     ecx, keys2
        and     ecx, 0ffffh
        or      ecx, 2
        mov     eax, ecx
        xor     ecx, 1
        xor     edx, edx
        mul     ecx
        shr     eax, 8
        push    eax
        invoke
ZipUpdateKeys, a
        pop     eax
        xor     al, a

        ret
ZipEncode endp
```

```
; Encode a single byte
ZipEncode proc a: BYTE
        push    ebx
        mov     ecx, keys2
        and     ecx, 0ffffh
        or      ecx, 2
        mov     ebx, ecx
        xor     ecx, 1
        xor     edx, edx
        mul     ecx
        shr     ebx, 8
        push    ebx
        invoke
ZipUpdateKeys, a
        pop     ebx
        xor     al, a
        pop     ebx
        ret
ZipEncode endp
```

EBX (Stack Base Register)를 General Purpose Register로 사용하기 위해 원본 값을 Push하여 저장해 둠

ECX 값을 EBX로 복사 (MOV EBX ECX)

ECX를 General Purpose Register로 간주하여, 연산의 Operand로 사용

외부 호출(invoke)시, EBX 값 보존을 위해 메모리에 저장해 둠 (PUSH EBX)

외부 호출(invoke) 종료 후, 저장했던 값을 뽑아내어 다시 사용함 (POP EBX)

동일한 의미의 다른 수식 (Semantic Obfuscation)

http://usecurity.hanyang.ac.kr

- 동일한 의미의 다른 수식(명령어)를 이용하여 코드의 내용을 다르게 보이도록 만드는 난독화 방법
- xor eax, eax는 항상 eax를 0으로 만든다. (초기화 코드) 즉 mov eax, 0과 동일한 의미를 가진다.

<pre>; Encode a single byte
ZipEncode proc a: BYTE
        mov   ecx, keys2
        and   ecx, 0ffffh
        or   ecx, 2
        mov   eax, ecx
        xor   ecx, 1
        xor   edx, edx
        mul   ecx
        shr   eax, 8
        push   eax
        invoke
ZipUpdateKeys, a
        pop   eax
        xor   al, a
        ret
ZipEncode endp</pre>
	<pre>; Encode a single byte
	ZipEncode proc a: BYTE
	        mov ecx, keys2
	        and ecx, 0ffffh
	        or ecx, 2
	        mov eax, ecx
	        xor ecx, 1
	        mov edx, 0
	        mul ecx
	        shr eax, 8
	        push eax
	        invoke
	ZipUpdateKeys, a
	        pop eax
	        xor al, a
	        ret
	ZipEncode endp</pre>

XOR EDX, EDX는 항상 EDX를 0으로 만든다.
즉 MOV EDX, 0과 동일한 의미를 가진다.

또한 SUB EDX, EDX 또는
DIV EDX, EDX; DEC EDX; 등
다른 수식으로도 얼마든지 대체 가능하다.