Compiler Construction

Chapter 7: Code Shape

Dittaya Wanvarie

Department of Mathematics and Computer Science
Chulalongkorn University

Second semester, 2024

Code shape

Translation choices

- Runtime speed
- Memory requirement
- Register optimization

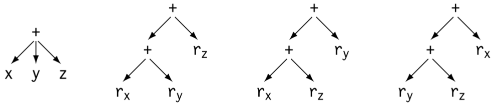
For example, how should we translate the `switch` statement in C?

- A series of if-else
- Array access
- Hashing

Translation choices

	Source Code	Low-Level, Three-Address Code		
Code	$x + y + z$	$r_1 \leftarrow r_x + r_y$ $r_2 \leftarrow r_1 + r_z$	$r_1 \leftarrow r_x + r_z$ $r_2 \leftarrow r_1 + r_y$	$r_1 \leftarrow r_y + r_z$ $r_2 \leftarrow r_1 + r_x$

Tree



■ **FIGURE 7.1** Alternate Code Shapes for $x + y + z$.

How should we translate `x+y+z` into 3-address codes?

- `r1 = rx + ry; r2 = r1 + rz;`
- `r1 = rx + rz; r2 = r1 + ry;`
- `r1 = ry + rz; r2 = r1 + rx`

What if the expression is `x+2+3`?

Arithmetic operators

- Base-offset for variables
- Immediate for constants
- Function call
- Automatic type conversion
- Assignment

Offset

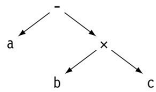
Given an activation record with the base address in `r_arp`, we may load variable `a` into a temporary register `r_a` by

```
loadI    @a           -> r_1
loadAO   r_arp, r_1 -> r_a
```

Arithmetic operators

```
expr(node) {
    int result, t1, t2;
    switch(type(node)) {
        case ×, ÷, +, -:
            t1 ← expr(LeftChild(node));
            t2 ← expr(RightChild(node));
            result ← NextRegister();
            emit(op(node), t1, t2, result);
            break;
        case NAME:
            entry  ← STLookup(node);
            result ← ValueIntoReg(entry);
            break;
        case NUMBER:
            num    ← NumberFromNode(node);
            result ← NumberIntoReg(num);
            break;
    }
    return result;
}
```

(a) Treewalk Code Generator



(b) Abstract Syntax Tree for
a - b x c

loadI	@a	$\Rightarrow$	r_1
loadAO	r_{arp}, r_1	$\Rightarrow$	r_2
loadI	@b	$\Rightarrow$	r_3
loadAO	r_{arp}, r_3	$\Rightarrow$	r_4
loadI	@c	$\Rightarrow$	r_5
loadAO	r_{arp}, r_5	$\Rightarrow$	r_6
mult	r_4, r_6	$\Rightarrow$	r_7
sub	r_2, r_7	$\Rightarrow$	r_8

(c) Naive Code

■ **FIGURE 7.2** Simple Treewalk Code Generator for Expressions.

Other concerns in arithmetic operations

Commutativity, associativity, and number systems

- Common subexpression help improving the code quality
- However, floating-point operations should NOT be re-ordered

Function calls in an expression

- If the return value is put in a register,
- The change in evaluation order may have side effects to the call

Solution: activation record

Mixed-type expressions

- The compiler must insert the conversion code

+	int	float
int	int	float
float	float	float

Conversion Table for +

Assignment operator

1. Evaluate the right-hand side of the assignment to a **value**
2. Evaluate the left-hand side of the assignment to a **location**
3. Store the right-hand side value into the left-hand side location

- R-value: an expression is evaluated to a **value**
- L-value: an expression is evaluated to a **location**

Reducing demand for registers

loadI	@a	⇒	r_1
loadAO	r_{arp}, r_1	⇒	r_2
loadI	@b	⇒	r_3
loadAO	r_{arp}, r_3	⇒	r_4
loadI	@c	⇒	r_5
loadAO	r_{arp}, r_5	⇒	r_6
mult	r_4, r_6	⇒	r_7
sub	r_2, r_7	⇒	r_8

(a) Code from Fig. 7.2(c)

loadI	@a	⇒	r_1
loadAO	r_{arp}, r_1	⇒	r_1
loadI	@b	⇒	r_2
loadAO	r_{arp}, r_2	⇒	r_2
loadI	@c	⇒	r_3
loadAO	r_{arp}, r_3	⇒	r_3
mult	r_2, r_3	⇒	r_2
sub	r_1, r_2	⇒	r_2

(b) Code After Register Allocation

loadI	@c	⇒	r_1
loadAO	r_{arp}, r_1	⇒	r_2
loadI	@b	⇒	r_3
loadAO	r_{arp}, r_3	⇒	r_4
mult	r_2, r_4	⇒	r_5
loadI	@a	⇒	r_6
loadAO	r_{arp}, r_6	⇒	r_7
sub	r_7, r_5	⇒	r_8

(c) Evaluate $b \times c$ First

loadI	@c	⇒	r_1
loadAO	r_{arp}, r_1	⇒	r_1
loadI	@b	⇒	r_2
loadAO	r_{arp}, r_2	⇒	r_2
mult	r_1, r_2	⇒	r_1
loadI	@a	⇒	r_2
loadAO	r_{arp}, r_2	⇒	r_2
sub	r_2, r_1	⇒	r_1

(d) Code After Register Allocation

■ **FIGURE 7.3** Rewriting a - b x c to Reduce Demand for Registers.

Accessing parameter values

Call-by-value

- Similar to local variable, using AR

Call-by-reference

- Save the address (pointer) into the AR. The retrieval needs 2 dereferencing steps
- Passing parameter d

```
loadI   @d          -> r1
loadAO  r_arp, r_1  -> r_2
load    r_2         -> r3
```

Access methods for values

- Variables stored in a register
- Variables stored in memory: `base + offset`
- Local variables: `r_arp + offset`
- Local variables of surrounding scopes: level information e.g. from access links/global display
- Static and global variables: addresses based on labels e.g. `loadI <label> -> r_i`
- Variables passed as parameters
- Variables stored in the heap
- Access methods for aggregates e.g. objects, structs, vectors, strings

Structure

- C's **`struct`**
- Pascal's **`record`**
- Object's record

Structure layouts

Linkedlist

Structure Layout Table

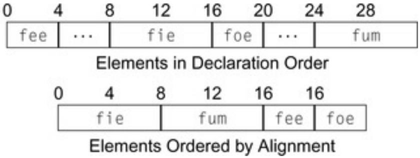
Name	Length	1^{st} Element
node	8	•
...	...	•

Structure Element Table

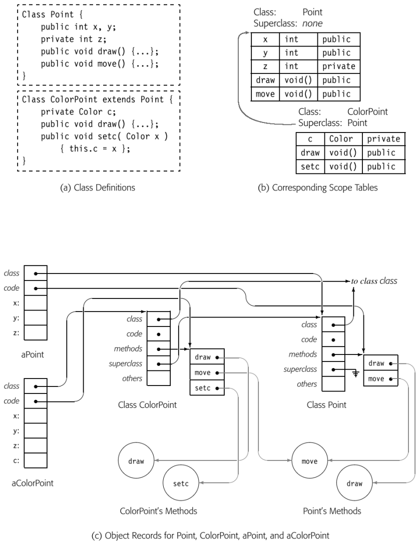
Name	Length	Offset	Type	Next
node.value	4	0	int	•
node.next	4	4	struct node *	•
...	...	...	...	...

Structure layout

```
struct example {
  int fee;
  double fie;
  int foe;
  double fum;
};
```

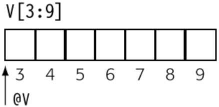


Object reference



■ **FIGURE 6.3** Object Layout, Linking, and Inheritance.

Vector addressing



Vector Layout

```
subI    r_i 3         r_1 // (i - lower bound)
multI   r_1 4         r_2 // x element length (4)
add     r_@v    r_2   r_3 // address of v[i]
load    r_3           r_v[i] // get the value of v[i]
```

- The false zero of a vector `v` is the address where `v[0]` would be

The location of `v[i]` is $(i - low) \times w$ where

- low is the lower bound
- w is the element length
- Then, `v[0] = v - low * w`

Storing and accessing arrays

Base and offset calculation

Example: accessing `a[4]` whose element in `a` requires 4 bytes

```
loadI   @a_0          r_a0    # base
multI   r_i,    4     r_2     # offset size
addI    r_a0,   r_2   r_3     # add offset
load    r_3           r_ai
```

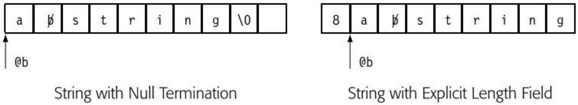
However, the the first index is not 0, we need to adjust the offset calculation

Multiplication to addition

If the offset size `w` in `r_i * w` is a power of two, we can use **shift** operation instead

- Fewer processing cycle
- E.g., `lshiftI r_i 2 -> r_2` is equivalent with `r_i * 4`

String representation



String assignment

```
a[1] = b[2];
```

Generated codes depend on the target machine instruction set

```
loadI  @b -> r_b
cloadI r_b, 2 -> r_2
loadI @a -> r_a
cstoreAI r_2 -> r_a, 1
```

Or

```
loadI  0x0000FF00  => r_c2     // mask for 2nd char
loadI  0xFF00FFFF  => r_c124   // mask for chars 1, 2, & 4
loadI     @b          => r_@b  // address of b
load      r_@b        => r_1   // get 1st word of b

and       r_1, r_c2   => r_2   // mask away others
lshiftI   r_2, 8      => r_3   // move it over 1 byte

loadI     @a          => r_@a  // address of a
load      r_@a        => r_4   // get 1st word of a

and       r_4, r_c124 => r_5   // mask away 2nd char
or        r_3, r_5    => r_6   // put in new 2nd char
store     r_6         => r_@a  // put it back in a
```

String assignment

a = b;

```
      loadI     @b        => r@b
      loadAI    r@b,-4    => r1          // get b's length
      loadI     @a        => r@a
      loadAI    r@a,-4    => r2          // get a's length
      cmp_LT    r2,r1     => r3          // will b fit in a?
      cbr       r3        -> Lsov,L1     // raise overflow

L1:   loadI     0         => r4          // counter
      cmp_LT    r4,r1     => r5          // more to copy?
      cbr       r5        -> L2,L3

L2:   cloadAO   r@b,r4    => r6          // get char from b
      cstoreAO  r6        => r@a,r4      // put it in a
      addI      r4,1      => r4          // increment offset
      cmp_LT    r4,r1     => r7          // more to copy?
      cbr       r7        -> L2,L3

L3:   storeAI   r1        => r@a,-4      // set length
```

String assignment

```
t1 = a;
t2 = b;
do {
  *t1++ = *t2++;
} while (*t2 != '\0')
```

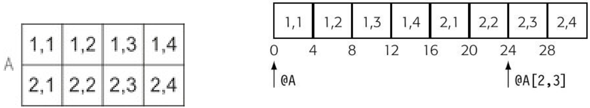
```
      loadI   @b      => r@b    // get pointers
      loadI   @a      => r@a
      loadI   NULL    => r1     // terminator
      cload   r@b     => r2     // get next char
L1:   cstore  r2      => r@a    // store it
      addI    r@b,1   => r@b    // bump pointers
      addI    r@a,1   => r@a
      cload   r@b     => r2     // get next char
      cmp_NE  r1,r2   => r4
      cbr     r4      -> L1,L2
L2:   nop                       // next statement
```

Other string operations

- String concatenation
- String length

Array storage layout

Row-major and column-major choices



Let

- `low1` be the first index of the first dimension
- `low2` be the first index of the second dimension
- `high1` be the first dimension's upper bound
- `high_2` be the second dimension's upper bound
- `len_k = high_k - low_k + 1` be the size of dimension `k`
- `w` be the size of each element

The location of `A[i,j]` is

`A + (i - low_1) * len_2 * w + (j - low_2) * w`

Array addressing

We can simplify

```
A + (i - low_1) * len_2 * w + (j - low_2) * w
```

into

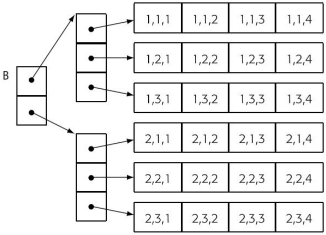
```
A + i * len_2 * w - low_1 * len_2 * w + j * w - low_2 * w
= A + (i * len_2 * w) + (j * w) - (low_1 * len_2 * 2 - low_2 * w)
= A + (i * len_2 * w) + (j * w) + A_0
= A_0 + (i * len_2 + j) * w
```

If `i, j` are in `r_i, r_j` then

```
loadI   A_0           -> r_A0 // false zero
multI   r_i    len_2  -> r_1  // i * len_2
add     r_1    r_j    -> r_2  // + j
multI   r_2    4      -> r_3  // *w, w = 4
loadAO  r_A0   r_3    -> r_A  // A[i,j]
```

High dimensional array

Indirection address



■ **FIGURE 7.4** Indirection Vectors in Row-Major Order for `B[1:2,1:3,1:4]`.

To access `B[i,j,k]`

```
loadI    B_0            -> r_B0 // false zero
multI    r_i     4      -> r_1  // pointer size = 4
loadAO   r_B0    r_1    -> r_2  // Address of B[i]
multI    r_j     4      -> r_3  // pointer size = 4
loadAO   r_2     r_3    -> r_4  // Address of B[i,j]
muliT    r_k     4      -> r_5  // w = 4
loadAO   r_4     r_5    -> r_b  // B[i,j,k]
```

Dope vectors

A descriptor for an actual parameter array

```
program main;
  begin;
    declare x(1:100,1:10,2:50),
      y(1:10,1:10,15:35) float;
    call fee(x);
    call fee(y);
  end main;
procedure fee(A)
  declare A(*,*,*) float;
      ...
  end fee;
```

(a) PL/I Code That Passes Whole Arrays

A →
@x_0
1
100
1
10
2
50

(b) Dope Vector for the First Call Site

A →
@y_0
1
10
1
10
15
35

(c) Dope Vector for the Second Call Site

■ **FIGURE 7.5** Dope Vectors.

Range check

A program that access an out-of-bound element is not well formed.

Naive range checking

```
for i in 1 .. n
    for j in 1 .. m
        if low_1 <= i <= high_1 and low_2 <= j <= high_2 then
            access a[i,j]
        else
            throw OutOfBoundException
```

Optimized version

```
if low_1 <= 1 <= and n <= high_1 and low_2 <= 1 and m <= high_2 then
    for i in 1 .. n
        for j in 1 .. m
            access a[i,j]
else
    throw OutOfBoundException
```

We can move the range check out of the two loops

Grammar

Adding boolean and relational operators to the expression grammar

$$
\begin{array}{lcl}
\mathit{Expr} & \rightarrow & \mathit{Expr} \vee \mathit{AndTerm} \\
 & | & \mathit{AndTerm} \\
\mathit{AndTerm} & \rightarrow & \mathit{AndTerm} \wedge \mathit{RelExpr} \\
 & | & \mathit{RelExpr} \\
\mathit{RelExpr} & \rightarrow & \mathit{RelExpr} < \mathit{NumExpr} \\
 & | & \mathit{RelExpr} \leq \mathit{NumExpr} \\
 & | & \mathit{RelExpr} = \mathit{NumExpr} \\
 & | & \mathit{RelExpr} \neq \mathit{NumExpr} \\
 & | & \mathit{RelExpr} \geq \mathit{NumExpr} \\
 & | & \mathit{RelExpr} > \mathit{NumExpr} \\
 & | & \mathit{NumExpr}
\end{array}
\qquad
\begin{array}{lcl}
\mathit{NumExpr} & \rightarrow & \mathit{NumExpr} + \mathit{Term} \\
 & | & \mathit{NumExpr} - \mathit{Term} \\
 & | & \mathit{Term} \\
\mathit{Term} & \rightarrow & \mathit{Term} \times \mathit{Value} \\
 & | & \mathit{Term} \div \mathit{Value} \\
 & | & \mathit{Factor} \\
\mathit{Value} & \rightarrow & \neg\ \mathit{Factor} \\
 & | & \mathit{Factor} \\
\mathit{Factor} & \rightarrow & (\mathit{Expr}) \\
 & | & \texttt{num} \\
 & | & \texttt{name}
\end{array}
$$

Representation

Numerical encoding: e.g. true (non-zero) and false (zero)

- If all variables are booleans, we may directly translate the operations into 3-address code operations.

  E.g. `b || c && ~d`

```
not r_d -> r_1
and r_c, r1 -> r_2
or r_b, r_2 -> r3
```

- Codes for conditional branch is quite messy

  E.g. `a < b`

```
    comp   r_a, r_b -> cc1
    cbr_LT cc1      -> L1, L2
L1: loadI  true     -> r_1
    jumpI           -> L3
L2: loadI  false    -> r_1
    jumpI           -> L3
L3: nop
```

Positional encoding

- Instead of storing the true and false values, we set the branch point for the true/false cases

E.g. `a < b || c < d && e < f`

```
    comp    ra, rb  ⇒ cc1    // a < b
    cbr_LT  cc1     → L1, L2
L1: loadI   true    ⇒ r1
    jumpI   → L3
L2: loadI   false   ⇒ r1
    jumpI   → L3

L3: comp    rc, rd  ⇒ cc2    // c < d
    cbr_LT  cc2     → L4, L5
L4: loadI   true    ⇒ r2
    jumpI   → L6
L5: loadI   false   ⇒ r2
    jumpI   → L6

L6: comp    re, rf  ⇒ cc3    // e < f
    cbr_LT  cc3     → L7, L8
L7: loadI   true    ⇒ r3
    jumpI           → L9
L8: loadI   false   ⇒ r3
    jumpI           → L9

L9: and     r2, r3  ⇒ r4
    or      r1, r4  ⇒ r5
```

(a) Naive Encoding

```
    comp    ra, rb  ⇒ cc1    // a < b
    cbr_LT  cc1     → L3, L1

L1: comp    rc, rd  ⇒ cc2    // c < d
    cbr_LT  cc2     → L2, L4

L2: comp    re, rf  ⇒ cc3    // e < f
    cbr_LT  cc3     → L3, L4

L3: loadI   true    ⇒ r5
    jumpI           → L5

L4: loadI   false   ⇒ r5
    jumpI           → L5

L5: nop
```

(b) Positional Encoding with Short-Circuit Evaluation

Hardware support

Relational operations

- A set of comparison operations
  - $<\leq, =, \geq, >, \neq$
  - `cmp_EQ r_a, r_b -> r_c`
- A set of operations that interpret the result of the comparison
  - `cbr r_c -> L_T, L_F`

Simple `if-else` implementation

Original code

```
if (a < b) then
    stmt1
else
    stmt2
```

Translation

```
        cmp_LT  r_a,  r_b   -> r_1
        cbr     r_1         -> L_1, L_2
L_1:    stmt1
        jumpI   L_3
L_2:    stmt2
        jumpI   L_3
L_3:    nop
```

Boolean expression implementation

Original code

```
a < b and c > d
```

Translation

```
cmp_LT  r_a,    r_b -> r_1
cmp_GT  r_c,    r_d -> r_2
and     r_1,    r_2 -> r_3
cbr     r_3         -> L_1, L_2
cbr     r_1         -> L_1, L_2
L_1:    stmt1                   # True
jumpI   L_3
L_2:    stmt2                   # False
jumpI   L_3
L_3:    nop
```

Short-circuit evaluation

Boolean identities

- `a or True == True`
- `a and False == False`

Original code

```
x = a < b or c < d and e < f
```

Naive translation

```
cmp_LT  r_a,    r_b -> r_1
cmp_LT  r_c,    r_d -> r_2
cmp_LT  r_e,    r_f -> r_3
and     r_2,    r_3 -> r_4
or      r-1,    r_4 -> r_x
```

Short-circuit evaluation

Original code

```
x = a < b or c < d and e < f
```

Short-circuit implementation

```
        cmp_LT  r_a,    r_b -> r_x
        cbr     r_x         -> L_3, L_1
L_1:    cmp_LT  r_c,    r_d -> r_x
        cbr     r_x         -> L_2, L_3
L_2:    cmp_LT  r_e,    r_f -> r_x
        jumpI               -> L_3
L_3:    nop
```

Variation in hardware support

There are 3 common variations

- Conditional move operations
- Predicated execution
- Conditional codes

Conditional move operations

- Take two inputs and assign one of them to the result, based on a boolean value or condition code (in another register)
- If the condition is an argument, the result instruction will be a 4-address code instruction
  - condition, source1, source2, target

Conditional move operations

Original code

```
if a < b then
    x = c + d
else
    x = e + f
```

Conditional move translation

```
add     r_c,    r_d     -> r_1
add     r_e,    r_f     -> r_2
cmpLT   r_a,    r_b     -> r_3
c_i2i   r_3, r_1, r_2   -> r_x
```

Note that there is no branching instruction

- If addition takes fewer cycles than the branch, conditional move will save runtime cycles

Predicated execution

- An operation will take effect if the predicate is true
- The predicate is then another argument, therefore it will be a 4-address code instruction
    - predicate, source1, source2, result

Predicated execution

Original code

```
if a < b then
    x = c + d
else
    x = e + f
```

Predicated execution

```
        cmpLT  r_a,  r_b   -> r_1
        not    r_1         -> r_2
(r_1)?  add    r_c,  r_d   -> r_x
(r_2)?  add    r_e,  r_f   -> r_x
```

- The processor may evaluate both expression and assign the final value based on the predicate
- Or, the processor may execute the operation only when the predicate is true

Condition codes

In stead of a boolean value (`true, false`), the comparison may return a condition code

- Each comparison result is set in a separate bit in the condition code, usually a special register
- This scheme requires a conditional branch based on the value in the condition code

Condition codes

Original code

```
a < b
```

Conditional codes translation

```
        comp    r_a,    r_b     -> cc
        cbr_LT  cc              -> L_1, L_2
L_1:    loadI   true            -> r_1
        jumpI                   -> L_3
L_2:    loadI   false           -> r_1
        jumpI                   -> L_3
L_3:    nop
```

- In the condition codes scheme, we have to explicitly load the boolean result to the target

Condition codes

Simple if-else construct

```
if (a < b) then
    stmt1
else
    stmt2
```

Naive translated code

```
        comp    r_a,    r_b     -> cc
        cbr_LT  cc              -> L_1, L_2
L_1:    loadI   true            -> r_1
        jumpI                   -> L_3
L_2:    loadI   false           -> r_1
        jumpI                   -> L_3
L_3:    load_I  true            -> r_2
        comp    r_1,    r_2     -> cc
        cbr     cc              -> L_4, L_5
L_4:    stmt1
        jumpI                   -> L_6
L_5:    stmt2
        jumpI                   -> L_6
L_6:    nop
```

Condition codes with optimization

Original code

```
if (a < b) then
    stmt1
else
    stmt2
```

Optimized translated code

```
        comp    r_a,    r_b     -> cc
        cbr_LT  cc              -> L_1, L_2
L_1:    stmt1
        jumpI                   -> L_3
L_2:    stmt2
        jumpI                   -> L_3
L_3:    nop
```

The implicit representation of `a < b` is in the condition code

Boolean code shape

Source Code	if (x < y) then a ← c + d else a ← e + f	
ILOC Code	comp $r_x, r_y \Rightarrow cc_1$ cbr_LT $cc_1 \rightarrow L_1, L_2$ L_1: add $r_c, r_d \Rightarrow r_a$ jumpI $\rightarrow L_{out}$ L_2: add $r_e, r_f \Rightarrow r_a$ jumpI $\rightarrow L_{out}$ L_{out}: nop Straight Condition Codes	cmp_LT $r_x, r_y \Rightarrow r_1$ cbr $r_1 \rightarrow L_1, L_2$ L_1: add $r_c, r_d \Rightarrow r_a$ jumpI $\rightarrow L_{out}$ L_2: add $r_e, r_f \Rightarrow r_a$ jumpI $\rightarrow L_{out}$ L_{out}: nop Boolean Compare
	comp $r_x, r_y \Rightarrow cc_1$ add $r_c, r_d \Rightarrow r_1$ add $r_e, r_f \Rightarrow r_2$ i2i_LT $cc_1, r_1, r_2 \Rightarrow r_a$ Conditional Move	cmp_LT $r_x, r_y \Rightarrow r_1$ not $r_1 \Rightarrow r_2$ (r_1)? add $r_c, r_d \Rightarrow r_a$ (r_2)? add $r_e, r_f \Rightarrow r_a$ Predicated Execution

(a) Using a Relational Expression to Govern Control Flow

Source Code	x ← a < b ∧ c < d	
ILOC Code	comp $r_a, r_b \Rightarrow cc_1$ cbr_LT $cc_1 \rightarrow L_1, L_2$ L_1: comp $r_c, r_d \Rightarrow cc_2$ cbr_LT $cc_2 \rightarrow L_3, L_2$ L_2: loadI false $\Rightarrow r_x$ jumpI $\rightarrow L_{out}$ L_3: loadI true $\Rightarrow r_x$ jumpI $\rightarrow L_{out}$ L_{out}: nop Straight Condition Codes	comp $r_a, r_b \Rightarrow cc_1$ i2i_LT $cc_1, r_T, r_F \Rightarrow r_1$ comp $r_c, r_d \Rightarrow cc_2$ i2i_LT $cc_2, r_T, r_F \Rightarrow r_2$ and $r_1, r_2 \Rightarrow r_x$ Conditional Move
		cmp_LT $r_a, r_b \Rightarrow r_1$ cmp_LT $r_c, r_d \Rightarrow r_2$ and $r_1, r_2 \Rightarrow r_x$ Boolean Compare
		cmp_LT $r_a, r_b \Rightarrow r_1$ cmp_LT $r_c, r_d \Rightarrow r_2$ and $r_1, r_2 \Rightarrow r_x$ Predicated Execution

(b) Using a Relational Expression to Produce a Value

Control-flow construct translation

- Building a representation for each block
- Connecting blocks with labels, branches, jumps

Basic building block

- Consecutive, **unlabeled, unpredicated** operations
  - Labeled statement might be the target of another goto statement
  - Predicated statement is the beginning of the goto statement
- Simple translation

Conditional execution

If statements

- The amount of code in the condition, then-block and else-block have great effect in the translation choice

Unit 1		Unit 2	
comparison $\Rightarrow$ r1			
(r1)	op1	(¬r1)	op11
(r1)	op2	(¬r1)	op12
(r1)	op3	(¬r1)	op13
(r1)	op4	(¬r1)	op14
(r1)	op5	(¬r1)	op15
(r1)	op6	(¬r1)	op16
(r1)	op7	(¬r1)	op17
(r1)	op8	(¬r1)	op18
(r1)	op9	(¬r1)	op19
(r1)	op10	(¬r1)	op20

(a) Using Predicates

	Unit 1	Unit 2
compare & branch		
L1:	op1	op2
	op3	op4
	op5	op6
	op7	op8
	op9	op10
	jumpI	→ L3
L2:	op11	op12
	op13	op14
	op15	op16
	op17	op18
	op19	op20
	jumpI	→ L3
L3:	nop	

(b) Using Branches

Conditional execution

Factors on choosing translation strategy

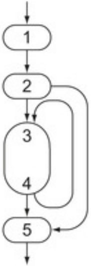
- Expected frequency of execution
- Uneven amounts of code
- Control flow inside the construct

Loops and iteration

General schema for a loop

```
For (e1: e2: e3) {
    loop body
}
```

(a) Example Code for Loop



Step	Purpose
1	Evaluate e_1
2	If $(\neg e_2)$ Then goto 5
3	*Loop Body*
4	Evaluate e_3 If (e_2) Then goto 3
5	*Code After Loop*

(b) Schema for Implementing Loop

For loops

Original code

```
for (i=1; i <= 100; i) {
    loop body
}
next statement
```

Translated codes

```
        loadI   1          -> r_1
        loadI   100        -> _r_1
        cmp_GT  r_i,    r_1 -> r_2
        cbr     r_2        -> L_2, L_1
L_1:    loop body
        addI    r_i     1   -> r_i
        cmp_LE  r_i,    r_1 -> r_3
        cbr     r_3        -> L_1, L_2
L_2:    next statment
```

Do loops

Original code

```
    do  10 i = 1, 100, 1
        loop body
10      continue
next statement
```

Translated codes

```
        loadI   1             -> r_1
        loadI   100           -> _r_1
        cmp_GT  r_i,     r_1 -> r_2
        cbr     r_2           -> L_2, L_1
L_1:    loop body
        addI    r_i      1   -> r_i
        cmp_LE  r_i,     r_1 -> r_3
        cbr     r_3           -> L_1, L_2
L_2:    next statment
```

While loops

Original code

```
while (x < y) {
    loop body
}
next statement
```

Translated code

```
        cmp_GE  r_x,    r_y -> r_1
        cbr     r_1         -> L_1, L_1
L_1:    loop body
        cmp_LT  r_x,    r_y -> r_2
        cbr     r_2         -> L_1, L_2
L_2:    next statement
```

Until loops

Original code

```
{
    loop body
} until (x < y)
next statement
```

Translated code

```
L_1:    loop body
        cmp_LT  r_x,        r_y -> r_2
        cbr     r_2         -> L_2, L_1
L_2:    next statement
```

Other loop elements

Tail recursion

- Instead of translating into a function call, we may translate the tail recursion using loops

Break, skip, continue statement

- Jump to the target label

Case constructs

Linear search

```
switch (e1)  {
  case 0:   block0;
            break;
  case 1:   block1;
            break;
  case 3:   block3;
            break;
  default:  blockd;
            break;
}
```

(a) Switch Statement

```
t1 ← e1
if (t1 = 0)
   then block0
   else if (t1 = 1)
      then block1
      else if (t1 = 2)
         then block2
         else if (t1 = 3)
            then block3
            else blockd
```

(b) Implemented as a Linear Search

Case constructs

Directly computing the address

```
switch (e1)  {
   case 0:   block0
             break;
   case 1:   block1
             break;
   case 2:   block2
             break;
   ...
   case 9:   block9
             break;
   default:  blockd
             break;
}
```

(a) Switch Statement

Label
LB_0
LB_1
LB_2
LB_3
LB_4
LB_5
LB_6
LB_7
LB_8
LB_9

(b) Jump Table

```
t1 ← e1
if (0 > t1 or t1 > 9)
   then jump to LBd
   else
      t2 ←@Table + t1 x 4
      t3 ← memory(t2)
      jump to t3
```

(c) Code for Address Computation

Binary search

```
switch (e1) {
   case 0:  block0
            break;
   case 15: block15
            break;
   case 23: block23
            break;
   ...
   case 99: block99
            break;
   default: blockd
            break;
}
```

(a) Switch Statement

Value	Label
0	LB_0
15	LB_{15}
23	LB_{23}
37	LB_{37}
41	LB_{41}
50	LB_{50}
68	LB_{68}
72	LB_{72}
83	LB_{83}
99	LB_{99}

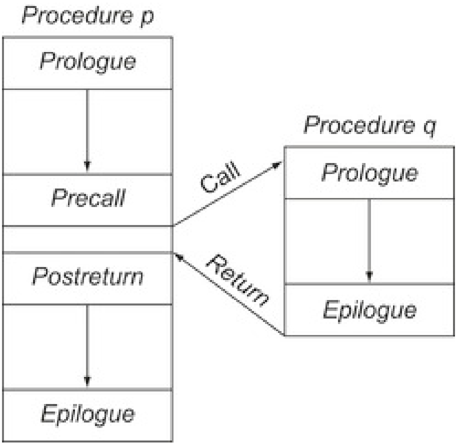
(b) Search Table

```
t1 ← e1
down ← 0   // lower bound
up ← 10    // upper bound + 1
while (down + 1 < up) {
   middle ← (up + down) ÷ 2
   if (Value [middle] ≤ t1)
      then down ← middle
      else up ← middle
}
if (Value [down] = t1
   then jump to Label[down]
   else jump to LBd
```

(c) Code for Binary Search

Procedure call

Linkage routine



Procedure call

If there are several call sites

- Moving precall and postreturn operations to prologue and epilogue will reduce the overall size of the translated codes

If there is only one call

- Moving procedure inline (i.e. no call) at the point of the call will reduce both the size and runtime cost

Passing parameters

Precall sequence

- Evaluate actual parameters to the call
- Pass values, or addresses to the location for that parameter
  - Either a register or callee's AR
- Pass implicit arguments
  - Caller's ARP, return addresses, addressability
  - Object record pointer (e.g. `this` in Java or `self` in Python)
- Pass a procedure as an argument
  - Location of the parameter
  - Access link information

Saving and restoring registers

- Caller-saves vs. callee-saves registers
- Standard library routines for register save/restore operations can save the code size
- Shared information between the caller and the callee

Some features on ISAs can reduce the code size or runtime speed

- Spill a portion of the register set
- Multiregister memory operations, e.g. double word, quad word operations

Translating using an attribute grammar

Write an attribute grammar with a syntax-driven translation for following control flow construct

- If-else
- while-loop

What are necessary attributes in the translation?

Example

Start with this grammar

$$
\begin{array}{lcl}
Expr & \rightarrow & Expr \vee AndTerm \\
 & | & AndTerm \\
AndTerm & \rightarrow & AndTerm \wedge RelExpr \\
 & | & RelExpr \\
RelExpr & \rightarrow & RelExpr < NumExpr \\
 & | & RelExpr \leq NumExpr \\
 & | & RelExpr = NumExpr \\
 & | & RelExpr \neq NumExpr \\
 & | & RelExpr \geq NumExpr \\
 & | & RelExpr > NumExpr \\
 & | & NumExpr
\end{array}
\qquad
\begin{array}{lcl}
NumExpr & \rightarrow & NumExpr + Term \\
 & | & NumExpr - Term \\
 & | & Term \\
Term & \rightarrow & Term \times Value \\
 & | & Term \div Value \\
 & | & Factor \\
Value & \rightarrow & \neg\ Factor \\
 & | & Factor \\
Factor & \rightarrow & (\,Expr\,) \\
 & | & \texttt{num} \\
 & | & \texttt{name}
\end{array}
$$

Example

(cont.)

$$
\begin{aligned}
Program &\rightarrow Block \\
Stmt &\rightarrow \textsf{name} = Expr \\
&\mid \textsf{if}\ RelExpr\ \textsf{then}\ Block \\
&\mid \textsf{if}\ RelExpr\ \textsf{then}\ WithElse\ Block \\
&\mid \textsf{while}\ RelExpr\ Block \\
&\mid \textsf{pass} \\
WithElse &\rightarrow \textsf{if}\ RelExpr\ \textsf{then}\ WithElse\ \textsf{else}\ WithElse \\
Block &\rightarrow Stmt \\
&\mid Stmt\ Block
\end{aligned}
$$

Example

From the following production

$$Stmt \to \textsf{while}\ RelExpr\ Block$$

We want to generate intermediate codes

```
label1: Codes for RelExpr
        Branch if false to label2
        Codes for Block
        Jump to label1
label2:
```

We may need following attributes

- $Stmt.next$ label
- $Stmt.code$
- $RelExpr.true$ label
- $RelExpr.false$ label
- $RelExpr.code$
- $RelExpr.result$

Example

Syntax-driven translation

$Stmt \rightarrow$ **while**

{

$L1 = newLabel()$
$L2 = newLabel()$
$RelExpr.false = Stmt.next$
$RelExpr.true = L2$

}

$RelExpr$ $\{ Block.next = L1 \}$

$Block$

{

$Stmt.code = printLabel(L1)$
$+RelExpr.code$
$+$cbr $RelExpr.result$->$RelExpr.true$,$RelExpr.false$
$+printLabel(RelExpr.True)$
$+Block.code$
$+printLabel(L2)$

}

Instead of concatenating pieces of codes, we can print codes on-the-fly to save memory spaces.