CSE504 Compiler Design
Intermediate Code Generation

YoungMin Kwon

Intermediate Code Generation

- Benefits of using machine-independent intermediate code
  - Retargeting is facilitated
  - Machine-independent code optimizer can be applied
- Intermediate Representations
  - Syntax tree, Postfix notation, **Three-address code**

Three-Address Code

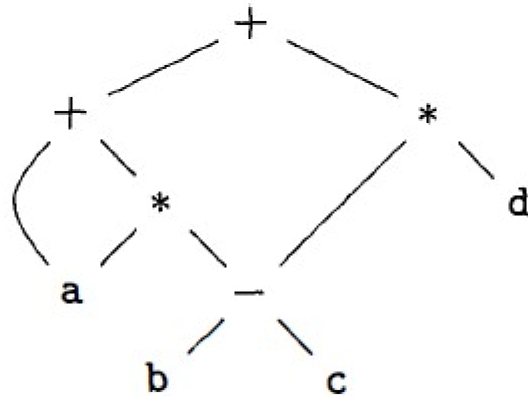
- Three-address code is a sequence of statements of the form

```
x := y op z
```

  - x, y, and z are names, constants, or compiler-generated temporaries
  - `x + y * z` can be translated into

```
t1 := y * z
t2 := x + t1
```

- Three-address code is a linearized representation of syntax trees or DAGs



(a) DAG

$t_1 = b - c$
$t_2 = a * t_1$
$t_3 = a + t_2$
$t_4 = t_1 * d$
$t_5 = t_3 + t_4$

(b) Three-address code

Three-Address Code

- Addresses
  - Name: source-program names
  - Constant: source-program constants
  - Compiler-generated temporary: create a distinct name each time a temporary is needed. Register-allocation will combine the temporaries.

Three Address Code Instructions

- Assignment
  - x = y op z
  - x = op y
- Copy
  - x = y
- Unconditional jump
  - jump: goto L
- Conditional jumps:
  - if x goto L
  - ifFalse x goto L
  - if x relop y goto L

Three Address Code Instructions

- Procedure calls
  - param x
  - call p, n
  - return y
- Indexed copy
  - x = y[i]
  - y[i] = x
- Address and pointer assignment
  - x = &y
  - x = *y
  - *x = y

param x_1
param x_2
...
param x_n
call p, n

Translation for Expression/Assignment

```
Production       Semantic Rules

S -> id := E     S.code := E.code || gen(id.place '=' E.place)

E -> E1 + E2     E.place := newtemp
                 E.code  := E1.code || E2.code ||
                            gen(E.place '=' E1.place '+' E2.place )

E -> E1 * E2     E.place := newtemp
                 E.code  := E1.code || E2.code ||
                            gen(E.place '=' E1.place '*' E2.place )

E -> - E1        E.place := newtemp
                 E.code  := E1.code ||
                            gen(E.place '=' 'minus' E1.place )

E -> ( E1 )      E.place := E1.place
                 E.code  := E1.code

E -> id          E.place := id.place
                 E.code  := ''
```

Quiz: Find the code for

`a := b + c * d`

Translation for While Statement

```
Production
S -> while E do S1

Semantic Rules
S.begin := newlabel
S.after := newlabel
S.code  := gen(S.begin) ||
           E.code ||
           gen('if' E.place '=' '0' 'goto' S.after) ||
           S1.code ||
           gen('goto' S.begin)
           gen(S.after ':')
```

Quiz: Translate

```
while a do
     a := a + b
```

Quadruples

- For a three-address code x = y + z, + in op, y in arg1, z in arg2, x in result
- Unary operators don't have arg2
- param doesn't have arg2 and result
- For jump operators, result is the target label

```
t1 = minus c
t2 = b * t1
t3 = minus c
t4 = b * t3
t5 = t2 + t4
 a = t5
```

(a) Three-address code

	op	*arg*$_1$	*arg*$_2$	*result*
0	minus	c		t_1
1	*	b	t_1	t_2
2	minus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a
		...		

(b) Quadruples

Declarations

```
P -> M D
M -> eps        { offset := 0 }

D -> D ; D
D -> id : T     { enter(id.name, T.type, offset)
                  offset := offset + T.width }

T -> integer    { T.type := integer, T.width := 4}
T -> real       { T.type := real,    T.width := 8}

T -> array [ num ] of T1
                { T.type  := array(num.val, T1.type)
                  T.width := num.val * T1.width }

T -> ^ T1       { T.type  := pointer(T1.type)
                  T.width := 4 }
```

Quiz: find the width and offset of each variable for the code

```
a : int ; b : real ; c : array [2] of real
```

Declarations: Nested Functions

```
P -> M D        { table.pop().addwidth(offset.pop()) }
M -> eps        { table.push(mktable(nil)), offset.push(0) }

D -> D ; D

D -> proc id ; N D1 ; S
                { t := table.pop(),
                  t.addwidth(offset.pop()),
                  table.top().enterproc(id.name, t) }
N -> eps        { table.push(mktable(table.top())),
                  offset.push(0) }

D -> id : T     { table.top().enter(
                        id.name, T.type, offset.top()),
                  offset.top() := offset.top() + T.width }
```

Quiz: find the width and offset of each variable for

```
a : int ;
proc f;
    b : int; c : real;
begin end;
d : real ;
```

Declarations: Records

- Records can be handled like nested procedures.

```
T -> record L D end
   { T.type  := record(table.pop()),
     T.width := offset.pop() }
L -> eps
   { table.push(mktable(nil)),
     offset.push(0) }
```

Assignment Statements

```
S -> id := E { p:= lookup(id.name),
               if p <> nil
               then emit( p '=' E.place)
               else error }
E -> E1 + E2 { E.place := newtemp,
               emit(E.place '=' E1.place '+' E2.place) }
E -> E1 * E2 { E.place := newtemp,
               emit(E.place '=' E1.place '*' E2.place) }
E -> - E1    { E.place := newtemp,
               emit(E.place '=' 'minus' E1.place) }
E -> ( E1 )  { E.place := E1.place }
E -> id      { p := lookup(id.name),
               if p <> nil
               then E.place := p
               else error }
```

Type Conversion (E ->E + E)

```
E.place = newtemp;
if E1.type = integer and E2.type = integer {
    emit(E.place '=' E1.place 'int+' E2.place);
    E.Type := integer
} else if E1.type = real and E2.type = real {
    emit(E.place '=' E1.place 'real+' E2.place);
    E.Type := real
} else if E1.type = integer and E2.type = real {
    u := newtemp;
    emit(u '=' 'inttoreal' E1.place);
    emit(E.place '=' u 'real+' E2.place);
    E.Type := real
} else if E1.type = real and E2.type = integer {
    u := newtemp;
    emit(u '=' 'inttoreal' E2.place);
    emit(E.place '=' E1.place 'real+' u);
    E.Type := real
} else
    E.Type = type_error
```

Quiz: write the translation for x := y + i * j when x, y are real, and i, j are integer

Boolean Expressions

- Two possible translations for boolean expressions
  - Encode true and false numerically (1: true, 0: false) and treat the expressions like arithmetic expressions.
  - Implementing a boolean expression by a flow of control.
    - E.g. E1 or E2: if E1 is true, the whole evaluation is true without evaluating E2.

Numerical Representation

```
E -> E1 or E2  { E.place := newtemp,
                 emit(E.place '=' E1.place 'or' E2.place) }
E -> E1 and E2 { E.place := newtemp,
                 emit(E.place '=' E1.place 'and' E2.place) }
E -> not E1    { E.place := newtemp,
                 emit(E.place '=' 'not' E1.place) }
E -> id1 op id2 {E.place := newtemp,
                 emit('if' id1.place op id2.place
                                      'goto' nextstat+3),
                 emit(E.place '=' 0),
                 emit('goto' nextstat+2),
                 emit(E.place '=' 1) }
E -> true      { E.place := newtemp,
                 emit(E.place '=' '1' }
E -> false     { E.place := newtemp,
                 emit(E.place '=' '0' }
```

Control-Flow Statements

```
S -> if E then S1
    E.true  := newlabel,
    E.false := S.next,
    S1.next := S.next,
    S.code  := E.code ||
               gen(E.true ':') ||
               S1.code
S -> if E then S1 else S2
    E.true  := newlabel,
    E.false := newlabel,
    S1.next := S.next,
    S2.next := S.next,
    S.code  := E.code ||
               gen(E.true ':') ||
               S1.code ||
               gen('goto' S.next) ||
               gen(E.false ':') ||
               S2.code
```

Control-Flow Statements

```
S -> while E do S1
    S.begin := newlabel,
    E.true  := newlabel,
    E.false := S.next,
    S1.next := S.begin
    S.code  := gen(S.begin, ‘:’) ||
               E.code ||
               gen(E.true ‘:’) ||
               S1.code ||
               gen(‘goto’ S.begin)
```

Control-Flow Expressions

```
E -> E1 or E2
    E1.true  := E.true,
    E1.false := newlabel,
    E2.true  := E.true,
    E2.false := E.false,
    E.code   := E1.code ||
                gen(E1.false ':') ||
                E2.code
E -> E1 and E2 (Quiz)
E -> not E1
    E1.true  := E.false,
    E1.false := E.true,
    E.code   := E1.code
E -> id1 op id2
    E.code := gen('if' id1.place op od2.place
                    'goto' E.true) ||
              gen('goto' E.false)
E -> true  E.code := (Quiz)
E -> false E.code := (Quiz)
```

Quiz: translate

```
while a < b or a < c
do    a := b + c
```

Backpatching

- Two pass implementation of boolean expressions
  - Find labels
  - Generate code
- Single pass implementation
  - Issue: labels are not available when generating the code
- Backpatching
  - Generate branching statements without labels
  - Manage lists of goto statements that will get the same future labels
  - Fill in labels as they are determined

Backpatching

- Helper functions
  - **makelist(i)**: creates a new list containing i, an index into the array of quadruples.
  - **merge(p, q)**: concatenates lists p and q.
  - **backpatch(p, i)**: insert I as the target label for each jump statements in the list p.

Backpatching: boolean expr.

```
E -> E1 or M E2 {
    backpatch(E1.falselist, M.quad)
    E.truelist := merge(E1.truelist, E2.truelist)
    E.falselist := E2.falselist
}

M -> eps { M.quad = nextquad }

E -> E1 and M E2 {
    backpatch(E1.truelist, M.quad)
    E.truelist := E2.truelist
    E.falselist := merge(E1.falselist, E2.falselist)
}

E -> not E1 {
    E.truelist := E2.falselist
    E.falselist := E2.truelist
}
```

Backpatching: bool expr.

```
E -> id1 relop id2 {
    E.truelist  := makelist(nextquad)
    E.falselist := makelist(nextquad+1)
    emit('if' id1.place relop.op id2.place 'goto _')
    emit('goto _')
}

E -> true {
    E.truelist := makelist(nextquad)
    emit('goto _')
}

E -> false {
    E.falselist := makelist(nextquad)
    emit('goto _')
}

E -> ( E1 ) {
    E.truelist  := E1.truelist
    E.falselist := E1.falselist
}
```

Quiz: annotate attributes and translate

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

Backpatching: Flow-Control Stmts.

```
S -> if E then M1 S1 N else M2 S2 {
    backpatch(E.truelist, M1.quad)
    backpatch(E.falselist, M2.quad)
    S.nextlist := merge(S1.nextlist,
                        merge(N.nextlist, S2.nextlist))
}

N -> eps {
    N.nextlist := makelist(nextquad)
    emit('goto _')
}

M -> eps { M.quad := nextquad }

S -> if E then M S1 {
    backpatch(E.truelist, M.quad)
    S.nextlist := merge(E.falselist, S1.nextlist)
}
```

Quiz: Translate

```
if a < b or
   c < d and
   e < f
then a := b
else a := c
```

Backpatching: Flow-Control Stmts.

```
S -> while M1 E do M2 S1 {
    backpatch(S1.nextlist, M1.quad)
    backpatch(E.truelist, M2.quad)
    S.nextlist := E.falselist
    emit('goto' M1.quad)
}

S -> A { ... S.nextlist = nil }

S -> begin L end { S.nextist := L.nextlist }

L -> L1 ; M S {
    backpatch(L1.nextlist, M.quad)
    L.nextlist = S.nextlist
}

L -> S { L.nextlist := S.nextlist }
```

Backpatching

- Quiz: translate the code below while LR parsing

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

Procedure Calls

```
S -> call id ( Elist ) {
    for each item p on queue do
        emit('param' p);
    emit('call' id.place)
}

Elist -> Elist, E {
    queue.add(E.place)
}

Elist -> E {
    queue.empty()
    queie.add(E.place)
}
```