

Intermediate Code Generation

Part II

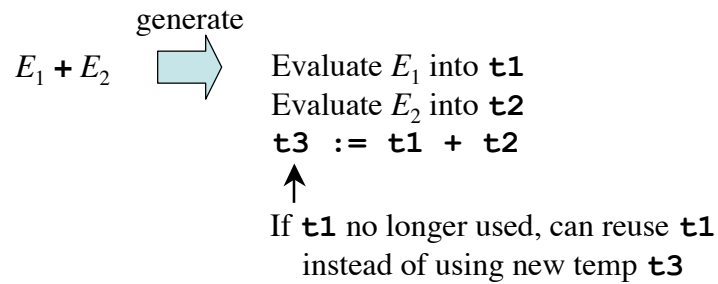
Chapter 8

COP5621 Compiler Construction
Copyright Robert van Engelen, Florida State University, 2005

Advanced Intermediate Code Generation Techniques

- Reusing temporary names
- Addressing array elements
- Translating logical and relational expressions
- Translating short-circuit Boolean expressions and flow-of-control statements with backpatching lists
- Translating procedure calls

Reusing Temporary Names



Modify *newtemp()* to use a “stack”:

Keep a counter *c*, initialized to 0

newtemp() increments *c* and returns temporary $\$c$

Decrement counter on each use of a $\$i$ in a three-address statement

Reusing Temporary Names (cont'd)

$x := a * b + c * d - e * f$



<i>Statement</i>	<i>c</i>
	0
$\\$0 := a * b$	1
$\\$1 := c * d$	2
$\\$0 := \\$0 + \\$1$	1
$\\$1 := e * f$	2
$\\$0 := \\$0 - \\$1$	1
$x := \\$0$	0

Addressing Array Elements: Multi-Dimensional Arrays

A : array [1..2,1..3] of integer; (Row-major)

$$\begin{aligned} \dots & := \mathbf{A}[i, j] = base_{\mathbf{A}} + ((i_1 - low_1) * n_2 + i_2 - low_2) * w \\ & = ((i_1 * n_2) + i_2) * w + c \end{aligned}$$



where $c = base_{\mathbf{A}} - ((low_1 * n_2) + low_2) * w$
with $low_1 = 1; low_2 = 1; n_2 = 3; w = 4$

```
t1 := i * 3
t1 := t1 + j
t2 := c // c = base_A - (1 * 3 + 1) * 4
t3 := t1 * 4
t4 := t2[t3]
... := t4
```

Addressing Array Elements: Grammar

$S \rightarrow L := E$

$E \rightarrow E + E$

| (E)

| L

$L \rightarrow Elist]$

| id

$Elist \rightarrow Elist , E$

| id [E

Synthesized attributes:

$E.place$	name of temp holding value of E
$Elist.array$	array name
$Elist.place$	name of temp holding index value
$Elist.ndim$	number of array dimensions
$L.place$	lvalue (=name of temp)
$L.offset$	index into array (=name of temp)
	null indicates non-array simple id

Addressing Array Elements

```

S → L := E   { if L.offset = null then
                emit(L.place ':=' E.place)
                else
                emit(L.place[L.offset] ':=' E.place) }
E → E1 + E2 { E.place := newtemp();
                emit(E.place ':=' E1.place '+' E2.place) }
E → ( E1 )   { E.place := E1.place }
E → L        { if L.offset = null then
                E.place := L.place
                else
                E.place := newtemp();
                emit(E.place ':=' L.place[L.offset] ) }

```


Addressing Array Elements

```


L → Elist ]   { L.place := newtemp();
                L.offset := newtemp();
                emit(L.place ':=' c(Elist.array);
                emit(L.offset ':=' Elist.place '*' width(Elist.array)) }
L → id        { L.place := id.place;
                L.offset := null }
Elist → Elist1, E
                { t := newtemp(); m := Elist1.ndim + 1;
                  emit(t ':=' Elist1.place '*' limit(Elist1.array, m));
                  emit(t ':=' t '+' E.place);
                  Elist.array := Elist1.array; Elist.place := t;
                  Elist.ndim := m }
Elist → id [ E { Elist.array := id.place; Elist.place := E.place;
                  Elist.ndim := 1 }

```

Translating Logical and Relational Expressions

`a or b and not c` 

```
t1 := not c
t2 := b and t1
t3 := a or t2
```

`a < b` 

```
if a < b goto L1
t1 := 0
goto L2
L1: t1 := 1
L2:
```

Translating Short-Circuit Expressions Using Backpatching

$E \rightarrow E \text{ or } M E$
 | $E \text{ and } M E$
 | **not** E
 | (E)
 | **id relop id**
 | **true**
 | **false**
 $M \rightarrow \epsilon$


Synthesized attributes:

$E.code$	three-address code
$E.truelist$	backpatch list for jumps on true
$E.falselist$	backpatch list for jumps on false
$M.quad$	location of current three-address quad

Backpatch Operations with Lists

- *makelist(i)* creates a new list containing three-address location *i*, returns a pointer to the list
- *merge(p₁, p₂)* concatenates lists pointed to by *p₁* and *p₂*, returns a pointer to the concatenated list
- *backpatch(p, i)* inserts *i* as the target label for each of the statements in the list pointed to by *p*


Backpatching with Lists: Example

a < b or c < d and e < f 

```

100: if a < b goto _
101: goto _
102: if c < d goto _
103: goto _
104: if e < f goto _
105: goto _

```

 backpatch

```

100: if a < b goto TRUE →
101: goto 102
102: if c < d goto 104
103: goto FALSE →
104: if e < f goto TRUE →
105: goto FALSE →

```

Backpatching with Lists: Translation Scheme

```

M → ε      { M.quad := nextquad() }
E → E1 or M E2
    { backpatch(E1.falselist, M.quad);
      E.truelist := merge(E1.truelist, E2.truelist);
      E.falselist := E2.falselist }
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 := E1.falselist;
              E.falselist := E1.truelist }
E → ( E1 ) { E.truelist := E1.truelist;
              E.falselist := E1.falselist }

```

Backpatching with Lists: Translation Scheme (cont'd)

```

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());
              E.falselist := nil;
              emit('goto _') }
E → false  { E.falselist := makelist(nextquad());
              E.truelist := nil;
              emit('goto _') }


```

Flow-of-Control Statements and Backpatching: Grammar


$S \rightarrow$ **if** E **then** S
 | **if** E **then** S **else** S
 | **while** E **do** S
 | **begin** L **end**
 | A
 $L \rightarrow$ $L ; S$
 | S

Synthesized attributes:

$S.nextlist$	backpatch list for jumps to the next statement after S (or nil)
$L.nextlist$	backpatch list for jumps to the next statement after L (or nil)

$S_1 ; S_2 ; S_3 ; S_4 ; S_4 \dots$ 

100: Code for S_1	$backpatch(S_1.nextlist, 200)$
200: Code for S_2	$backpatch(S_2.nextlist, 300)$
300: Code for S_3	$backpatch(S_3.nextlist, 400)$
400: Code for S_4	$backpatch(S_4.nextlist, 500)$
500: Code for S_5	

Jumps out of S_1 

Flow-of-Control Statements and Backpatching

$S \rightarrow A$ { $S.nextlist := nil$ }
 $S \rightarrow$ **begin** L **end**
 { $S.nextlist := L.nextlist$ }
 $S \rightarrow$ **if** E **then** $M S_1$
 { $backpatch(E.truelist, M.quad);$
 $S.nextlist := merge(E.falselist, S_1.nextlist)$ }
 $L \rightarrow L_1 ; M S$ { $backpatch(L_1.nextlist, M.quad);$
 $L.nextlist := S.nextlist;$ }
 $L \rightarrow S$ { $L.nextlist := S.nextlist;$ }
 $M \rightarrow \epsilon$ { $M.quad := nextquad()$ }

Flow-of-Control Statements and Backpatching (cont'd)

```

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)) }

S → while M1 E do M2 S1
    { backpatch(S1.nextlist, M1.quad);
      backpatch(E.truelist, M2.quad);
      S.nextlist := E.falselist;
      emit('goto _') }

N → ε
    { N.nextlist := makelist(nextquad());
      emit('goto _') }

```

Translating Procedure Calls

```

S → call id ( Elist )
Elist → Elist , E
       | E

```

```

foo(a+1, b, 7) →
t1 := a + 1
t2 := 7
param t1
param b
param t2
call foo 3

```

Translating Procedure Calls

$S \rightarrow \text{call id} (Elist)$ { **for** each item p on *queue* **do**
 $emit(\text{'param' } p);$
 $emit(\text{'call' id.place } |queue|)$ }

$Elist \rightarrow Elist , E$ { append $E.place$ to the end of *queue* }

$Elist \rightarrow E$ { initialize *queue* to contain only $E.place$ }