Intermediate Code Generation Part II

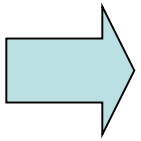
Chapter 8

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

generate

$E_1 + E_2$ 

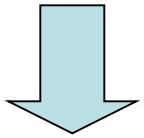
Evaluate E_1 into **`t1`**
Evaluate E_2 into **`t2`**
`t3 := t1 + t2`

If **`t1`** no longer used, can reuse **`t1`**
instead of using new temp **`t3`**

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



Statement	*c*
	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: One-Dimensional Arrays

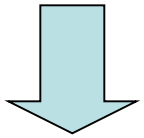
```
A : array [10..20] of integer;
```

… := A[i] $= base_A + (i - low) * w$

$= i * w + c$

where $c = base_A - low * w$

with $low = 10;\ w = 4$

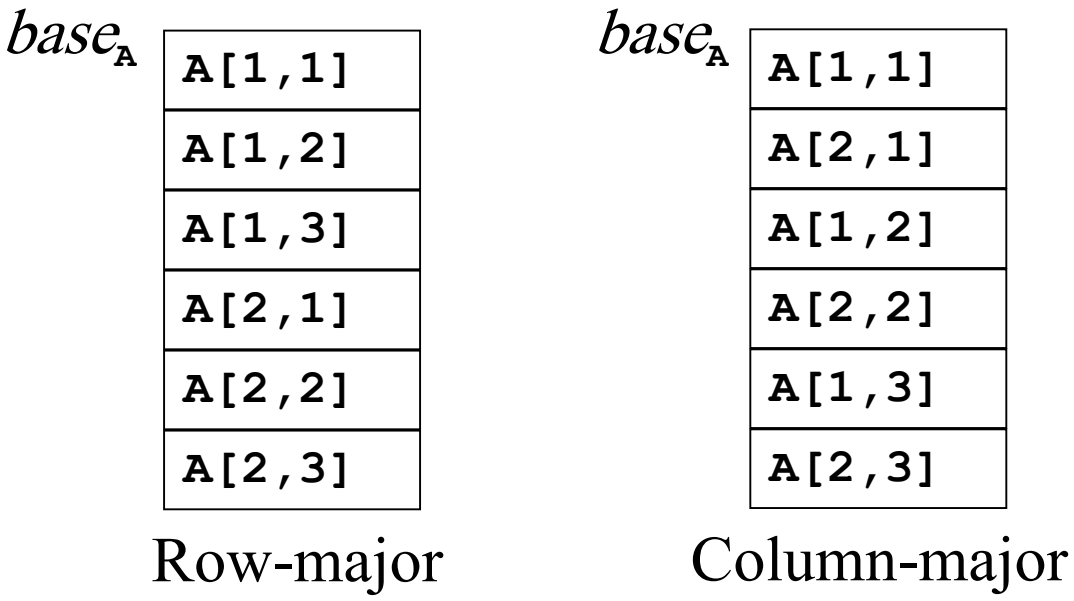


```
t1 := c        // c = baseA - 10 * 4
t2 := i * 4
t3 := t1[t2]
…  := t3
```

Addressing Array Elements: Multi-Dimensional Arrays

```
A : array [1..2,1..3] of integer;
```

$low_1 = 1,\ low_2 = 1,\ n_1 = 2,\ n_2 = 3,\ w = 4$



Addressing Array Elements: Multi-Dimensional Arrays

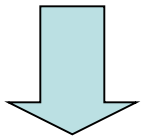
`A : array [1..2,1..3] of integer;` (Row-major)

… **`:= A[i,j]`** $= base_{\mathbf{A}} + ((i_1 - low_1) * n_2 + i_2 - low_2) * w$

$= ((i_1 * n_2) + i_2) * w + c$

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

$S \rightarrow L$ **:=** E { **if** L.offset = **null then**
emit(L.place ‘:=’ E.place)
else
emit(L.place[L.offset] ‘:=’ E.place) }

$E \rightarrow E_1$ + E_2 { E.place := *newtemp*();
emit(E.place ‘:=’ E_1.place ‘+’ E_2.place) }

$E \rightarrow$ **(** E_1 **)** { E.place := E_1.place }

$E \rightarrow L$ { **if** L.offset = **null then**
E.place := L.place
else
E.place := *newtemp*();
emit(E.place ‘:=’ L.place[L.offset] }

Addressing Array Elements

$L \rightarrow Elist$ **]** { *L*.place := *newtemp*();
L.offset := *newtemp*();
emit(*L*.place ‘:=’ *c*(*Elist*.array);
emit(*L*.offset ‘:=’ *Elist*.place ‘*’ *width*(*Elist*.array)) }

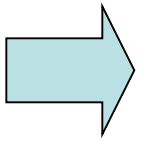
$L \rightarrow$ **id** { *L*.place := **id**.place;
L.offset := **null** }

$Elist \rightarrow Elist_1$ **,** *E*
{ *t* := *newtemp*(); *m* := $Elist_1$.ndim + 1;
emit(*t* ‘:=’ $Elist_1$.place ‘*’ *limit*($Elist_1$.array, *m*));
emit(*t* ‘:=’ *t* ‘+’ *E*.place);
Elist.array := $Elist_1$.array; *Elist*.place := *t*;
Elist.ndim := *m* }

Elist → **id [** *E* { *Elist*.array := **id**.place; *Elist*.place := *E*.place;
Elist.ndim := 1 }

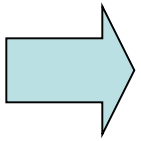
Translating Logical and Relational Expressions

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

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

$M \rightarrow \varepsilon$

Synthesized attributes:

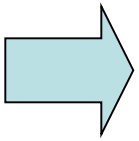
E.code	three-address code
E.truelist	backpatch list for jumps on true
E.falselist	backpatch list for jumps on false
M.quad	location of current three-address quad

Backpatch Operations with Lists

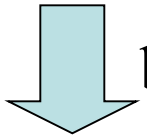
- *makelist*(*i*) creates a new list containing three-address location *i*, returns a pointer to the list
- *merge*(p_1, p_2) concatenates lists pointed to by p_1 and p_2, returns a pointer to the concatenates list
- *backpatch*(*p*, *i*) inserts *i* as the target label for each of the statements in 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

$M \rightarrow \varepsilon$ { M.quad := *nextquad*() }

$E \rightarrow E_1$ **or** $M\ E_2$
{ *backpatch*(E_1.falselist, M.quad);
E.truelist := *merge*(E_1.truelist, E_2.truelist);
E.falselist := E_2.falselist }

$E \rightarrow E_1$ **and** $M\ E_2$
{ *backpatch*(E_1.truelist, M.quad);
E.truelist := E_2.truelist;
E.falselist := *merge*(E_1.falselist, E_2.falselist); }

$E \rightarrow$ **not** E_1 { E.truelist := E_1.falselist;
E.falselist := E_1.truelist }

$E \rightarrow$ **(** E_1 **)** { E.truelist := E_1.truelist;
E.falselist := E_1.falselist }

Backpatching with Lists: Translation Scheme (cont'd)

$E \rightarrow$ **id**$_1$ **relop** **id**$_2$
{ *E*.truelist := *makelist*(*nextquad*());
E.falselist := *makelist*(*nextquad*() + 1);
emit('`if` ' **id**$_1$.place **relop**.op **id**$_2$.place '`goto _`');
emit('`goto _`') }

$E \rightarrow$ **true** { *E*.truelist := *makelist*(*nextquad*());
E.falselist := nil;
emit('`goto _`') }

$E \rightarrow$ **false** { *E*.falselist := *makelist*(*nextquad*());
E.truelist := nil;
emit('`goto _`') }

Flow-of-Control Statements and Backpatching: Grammar

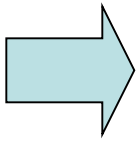
$S \rightarrow$ **if** E **then** S
| **if** E **then** S **else** S
| **while** E **do** S
| **begin** L **end**
| A

$L \rightarrow L$ **;** 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 …



```
100: Code for S1
200: Code for S2
300: Code for S3
400: Code for S4
500: Code for S5
```

Jumps out of S_1

backpatch(S_1.nextlist, 200)
backpatch(S_2.nextlist, 300)
backpatch(S_3.nextlist, 400)
backpatch(S_4.nextlist, 500)

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; }

$M \rightarrow \varepsilon$ { M.quad := *nextquad*() }

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

$S \rightarrow$ **if** E **then** M_1 S_1 N **else** M_2 S_2

{ *backpatch*(E.truelist, M_1.quad);
backpatch(E.falselist, M_2.quad);
S.nextlist := *merge*(S_1.nextlist,
merge(N.nextlist, S_2.nextlist)) }

$S \rightarrow$ **while** M_1 E **do** M_2 S_1

{ *backpatch*(S_1,nextlist, M_1.quad);
backpatch(E.truelist, M_2.quad);
S.nextlist := E.falselist;
emit('`goto _`') }

$N \rightarrow \varepsilon$ { N.nextlist := *makelist*(*nextquad*());
emit('`goto _`') }

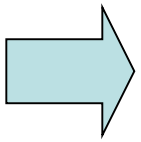
Translating Procedure Calls

$S \rightarrow$ **call id (** *Elist* **)**

$Elist \rightarrow Elist$ **,** E

$\mid E$

foo(a+1, b, 7)



```
t1 := a + 1
t2 := 7
param t1
param b
param t2
call foo 3
```

Translating Procedure Calls

$S \rightarrow$ **call id (** *Elist* **)**	{ **for** each item *p* on *queue* **do** *emit*(‘**`param`**’ *p*); *emit*(‘**`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 }