Translation Rules

```
S → if B then M S1
    { backpatch(B.truelist, M.quad)
      S.nextlist = mergelist(B.falselist, S1.nextlist)
    }
```

```
S → if B then M1 S1 N else M2 S2
    { backpatch(B.truelist, M1.quad)
      backpatch(B.falselist, M2.quad)
      S.nextlist = mergelist(S1.nextlist,
                        mergelist(N.nextlist, S2.nextlist))
    }
```

```
S → while M1 B do M2 S1
    { backpatch(S1.nextlist, M1.quad)
      backpatch(B.truelist, M2.quad)
      S.nextlist = B.falselist
      emit( 'goto' M1.quad)
    }
```

Translation Rules (Contd.)

S → begin L end
  { S.nextlist = L.nextlist }

S → A
  { S.nextlist = nil }

L → L1 M S
  { backpatch(L1.nextlist, M.quad)
   L.nextlist = S.nextlist
  }

L → S
  { L.nextlist = S.nextlist }

M → ε
  { M.quad = nextquad() }

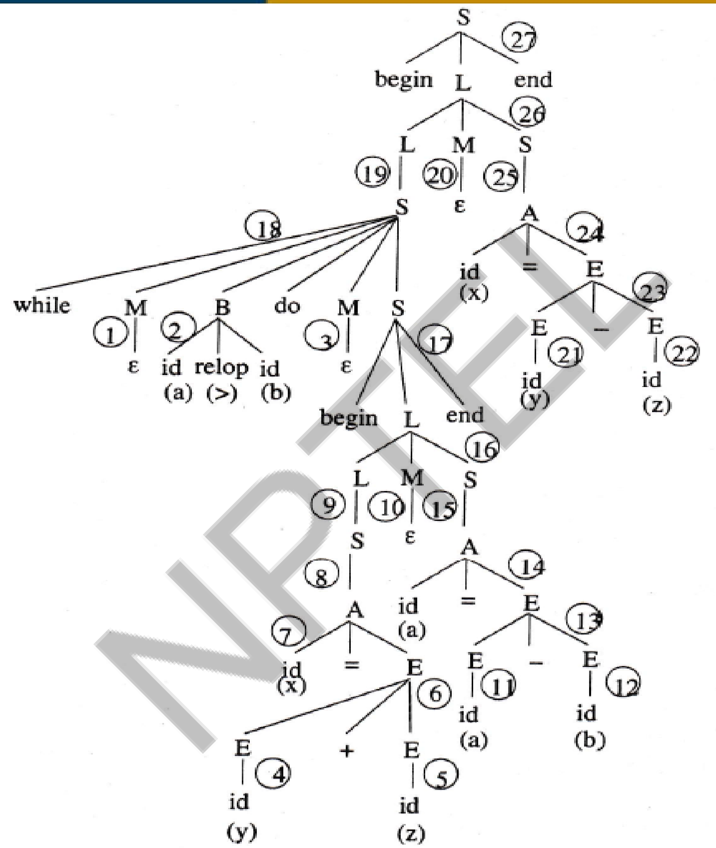
N → ε
  { N.nextlist = nextquad()
   emit('goto' ...)
  }

Example

```
begin
    while a > b do
    begin
        x = y + z
        a = a – b
    end
    x = y – z
end
```

Final Code:

```
1: if a > b goto 3
2: goto 8
3: t1 = y + z
4: x = t1
5: t2 = a – b
6: a = t2
7: goto 1
8: x = t3
```


S
27
begin
L
end
26
L
M
S
19
20
25
S
ε
A
18
24
id
(x)
=
E
23
while
M
B
do
M
S
1
2
3
17
ε
id
relop
id
(a)
(>)
(b)
ε
E
–
E
21
22
id
(y)
id
(z)
begin
L
end
16
L
M
S
9
10
15
S
ε
A
8
14
id
(a)
=
E
13
A
7
E
–
E
11
12
id
(a)
id
(b)
id
(x)
=
E
6
E
+
E
4
5
id
(y)
id
(z)

Red. no.	Action
1	$M.quad = 1$
2	$B.truelist = \{1\}$, $B.falselist = \{2\}$ Code generated: 1: if $a > b$ goto ... 2: goto ...
3	$M.quad = 3$
4	$E.place = y$
5	$E.place = z$
6	$E.place = t_1$ Code generated: 3: $t_1 = y + z$
7	Code generated: 4: $x = t_1$
8	$S.nextlist = \{\}$
9	$L.nextlist = \{\}$
10	$M.quad = 5$
11	$E.place = a$
12	$E.place = b$
13	$E.place = t_2$ Code generated: 5: $t_2 = a - b$
14	Code generated: 6: $a = t_2$
15	$S.nextlist = \{\}$
16	Backpatch($\{\}$, 5) $L.nextlist = \{\}$
17	$S.nextlist = \{\}$
18	backpatch($\{\}$,1) backpatch($\{1\}$,3) $\Rightarrow$ Code modified as: 1: if $a > b$ goto 3 $S.nextlist = \{2\}$ Code generated: 7: goto ...
19	$L.nextlist = \{2\}$
20	$M.quad = 8$
21	$E.place = y$
22	$E.place = z$
23	$E.place = t_3$ Code generated: 8: $t_3 = y - z$
24	Code generated: 9: $x = t_3$
25	$S.nextlist = \{\}$
26	Backpatch ($\{2\}$,8) $\Rightarrow$ Code modified as: 2: goto 8 $L.nextlist = \{\}$
27	$S.nextlist = \{\}$

Case Statements

```
switch(E) {
    case c1: ...
    ...
    case cn: ...
    default: ...
}
```

Implementation alternatives:

- Linear search for matching option
- Binary search for matching case
- A jump table
- Linear or binary search may be cheaper if number of cases small, for larger number of cases, jump table may be cheaper
- If case values are not clustered closely together, jump table may be too costly for space

Jump Table Implementation

Let the maximum and the minimum case values be c_{max} and c_{min} respectively

> Code to evaluate *E* into *t*
> if $t < c_{min}$ goto *Default_Case*
> if $t > c_{max}$ goto *Default_Case*
> goto *JumpTable*[*t*]
> *Default_Case*: ...

JumpTable[*i*] is the address of the code to execute, if *E* evaluates to *i*

Function Calls

- Can be divided into two subsequences
  - Calling sequence: set of actions executed at the time of calling a function
  - Return sequence: set of actions at the time of returning from the function call
- For both, some actions performed by Caller of the function and the other by the callee

Calling Sequence

Caller

- Evaluate actual parameters
- Place actuals where the callee wants them
- Corresponding three-address instruction:

  param t
- Save machine state (current stack and/or frame pointers, return address)
- Corresponding three-address instruction:

  call p, n (n=number of actuals)

Callee

- Save registers, if necessary
- Update stack and frame pointers to accommodate m bytes of local storage
- Corresponding three-address instruction:

  enter m

Return Sequence

Callee

- Place return value, if any, where the caller wants it
- Adjust stack/frame pointers
- Jump to return address
- Corresponding three-address instruction:

  return x or return

Caller

- Save the value returned by the callee
- Corresponding three-address instruction:

  retrieve x

Example Function Call

X = f(0, y+1) – 1

```
t1 = y + 1
param t1
param 0
call f, 2
retrieve t2
t3 = t2 – 1
x = t3
```

Storage Allocation for Functions

- Creates problem as the first instruction in a function is:

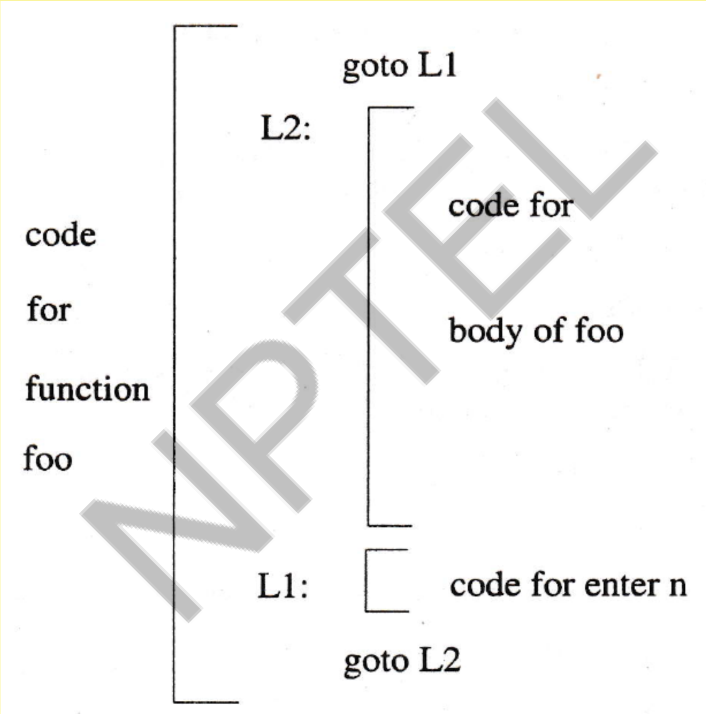
  enter n /* n = space for locals, temporaries */

- Value of n not known until the whole function has been processed.
- There can be two possible solutions
  - Generating final code in a list
  - Using pair of goto statements

Generating Final Code in List

- Generate final code in a list
- Backpatch the appropriate instructions after processing the function body
- Approach is similar to single-phase code generation for Boolean expressions and control flow statements
- Advantage: Possibility of machine dependent optimizations
- May be slow and may require more memory during code generation

Using Pair of goto Statements



Conclusion

- Intermediate code generation, though not mandatory, helps in retargeting the compiler towards different architectures
- Selecting a good intermediate language itself is a formidable task
- Three-address code is one such representation
- Syntax-directed schemes can be utilized to generate three-address code from the parse tree of the input program
- Translation of almost all major programming language constructs have been carried out

NPTEL ONLINE CERTIFICATION COURSES

Thank you!