Lecture 22-23

Intermediate Code Generation

Awanish Pandey

Department of Computer Science and Engineering
Indian Institute of Technology
Roorkee

March 29, 2025

Control flow translation of boolean expression

- E $\rightarrow$ not E_1 $\quad$ E_1.true $=$ E.false

Control flow translation of boolean expression

- E $\rightarrow$ not E_1 E_1.true = E.false
  E_1.false = E.true
  E.code = E_1.code

Control flow translation of boolean expression

- E $\rightarrow$ not E_1 $\quad$ E_1.true = E.false
  E_1.false = E.true
  E.code = E_1.code
- E $\rightarrow$ (E_1) $\quad$ E_1.true = E.true
  E_1.false = E.false

Control flow translation of boolean expression

- E $\rightarrow$ not E_1 $\quad$ E_1.true = E.false
  E_1.false = E.true
  E.code = E_1.code
- E $\rightarrow$ (E_1) $\quad$ E_1.true = E.true
  E_1.false = E.false
  E.code = E_1.code
- E $\rightarrow$ id_1 relop id_2

Control flow translation of boolean expression

- E $\rightarrow$ not E_1 $\quad$ E_1.true = E.false
  E_1.false = E.true
  E.code = E_1.code
- E $\rightarrow$ (E_1) $\quad$ E_1.true = E.true
  E_1.false = E.false
  E.code = E_1.code
- E $\rightarrow$ id_1 relop id_2
  E.code = gen(if id_1 relop id_2 goto E.true) || gen(goto E.false)

Control flow translation of boolean expression

- E $\rightarrow$ not E_1 $\quad$ E_1.true = E.false
  E_1.false = E.true
  E.code = E_1.code
- E $\rightarrow$ (E_1) $\quad$ E_1.true = E.true
  E_1.false = E.false
  E.code = E_1.code
- E $\rightarrow$ id_1 relop id_2
  E.code = gen(if id_1 relop id_2 goto E.true) || gen(goto E.false)
- E $\rightarrow$ `true` $\quad$ E.code = gen(goto E.true)

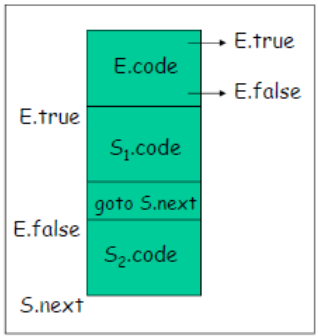
Control flow translation of boolean expression

- E $\rightarrow$ not E_1 $\quad$ E_1.true = E.false
  E_1.false = E.true
  E.code = E_1.code
- E $\rightarrow$ (E_1) $\quad$ E_1.true = E.true
  E_1.false = E.false
  E.code = E_1.code
- E $\rightarrow$ id_1 relop id_2
  E.code = gen(if id_1 relop id_2 goto E.true) || gen(goto E.false)
- E $\rightarrow$ `true` $\quad$ E.code = gen(goto E.true)
- E $\rightarrow$ `false` $\quad$ E.code = gen(goto E.false)

Control flow translation of boolean expression

- $E \rightarrow E_1$ or E_2 $\quad E_1.true = E.true$
  $E_1.false = \text{newlabel}()$
  $E_2.true = \text{E.true}$
  $E_2.false = \text{E.false}$
  $\text{E.code} = E_1.\text{code} \mid\mid \text{gen}(E_1.\text{false}) \mid\mid E_2.\text{code}$
- $E \rightarrow E_1$ and E_2 $\quad E_1.\text{true} = \text{newlabel}()$
  $E_1.\text{false} = \text{E.false}$
  $E_2.\text{true} = \text{E.true}$
  $E_2.\text{false} = \text{E.false}$
  $\text{E.code} = E_1.\text{code} \mid\mid \text{gen}(E_1.\text{true}) \mid\mid E_2.\text{code}$

If-else



$S \rightarrow$ if E then S_1 else S_2
$E.true = newlabel()$
$E.false = newlabel()$
$S_1.next = S.next$
$S_2.next = S.next$
$S.code = E.code||$
$gen(E.true \text{“ : ”})||$
$S_1.code||$
$gen(goto\ S.next)||$
$gen(E.false \text{“ : ”})||$
$S_2.code$

BackPatching

- Way to implement boolean expressions and flow of control statements in one pass

BackPatching

- Way to implement boolean expressions and flow of control statements in one pass
- Labels are indices into this array

BackPatching

- Way to implement boolean expressions and flow of control statements in one pass
- Labels are indices into this array
- **makelist(i):** create a newlist containing only i, return a pointer to the list.

BackPatching

- Way to implement boolean expressions and flow of control statements in one pass
- Labels are indices into this array
- **makelist(i):** create a newlist containing only i, return a pointer to the list.
- **merge(p1,p2):** merge lists pointed to by p1 and p2 and return a pointer to the concatenated list

BackPatching

- Way to implement boolean expressions and flow of control statements in one pass
- Labels are indices into this array
- **makelist(i):** create a newlist containing only i, return a pointer to the list.
- **merge(p1,p2):** merge lists pointed to by p1 and p2 and return a pointer to the concatenated list
- **backpatch(p,i):** insert i as the target label for the statements in the list pointed to by p

Boolean Expression

$E \rightarrow E_1$ or M E_2
E_1 and M E_2
$M \rightarrow \epsilon$

Boolean Expression

$E \rightarrow E_1$ or M E_2
E_1 and M E_2
$M \rightarrow \epsilon$

- Insert a marker non terminal M into the grammar to pick up index of next instruction.

Boolean Expression

$E \rightarrow E_1$ or M E_2
E_1 and M E_2
$M \rightarrow \epsilon$

- Insert a marker non terminal M into the grammar to pick up index of next instruction.
- Attributes `truelist` and `falselist` are used to generate jump code for boolean expressions

Boolean Expression

$E \rightarrow E_1$ or M E_2
E_1 and M E_2
$M \rightarrow \epsilon$

- Insert a marker non terminal M into the grammar to pick up index of next instruction.
- Attributes `truelist` and `falselist` are used to generate jump code for boolean expressions
- incomplete jumps are placed on lists pointed to by E.truelist and E.falselist

Boolean Expression

- Consider $E \rightarrow E_1$ and M E_2

Boolean Expression

- Consider $E \rightarrow E_1$ and M E_2
  - if E_1 is `false` then E is also `false` so statements in E_1.falselist become part of E.falselist.

Boolean Expression

- Consider $E \rightarrow E_1$ and M E_2
  - if E_1 is `false` then E is also `false` so statements in E_1.falselist become part of E.falselist.
  - if E_1 is `true` then E_2 must be tested so target of $E_1.truelist$ is beginning of E_2

Boolean Expression

- Consider $E \rightarrow E_1$ and M E_2
  - if E_1 is `false` then E is also `false` so statements in E_1.falselist become part of E.falselist.
  - if E_1 is `true` then E_2 must be tested so target of $E_1.truelist$ is beginning of E_2
  - target is obtained by marker M

Boolean Expression

- Consider $E \rightarrow E_1$ and M E_2
  - if E_1 is `false` then E is also `false` so statements in E_1.falselist become part of E.falselist.
  - if E_1 is `true` then E_2 must be tested so target of $E_1.truelist$ is beginning of E_2
  - target is obtained by marker M
  - attribute M.instr records the number of the first statement of E_2.code

Boolean Expression

E $\rightarrow$ id_1 relop id_2

E.truelist = makelist(nextinstr)

E.falselist = makelist(nextinstr+ 1)

emit(if id_1 relop id_2 goto —)

emit(goto —)

Boolean Expression

E $\rightarrow$ id_1 relop id_2
    E.truelist = makelist(nextinstr)
    E.falselist = makelist(nextinstr+ 1)
    emit(if id_1 relop id_2 goto —)
    emit(goto —)
E $\rightarrow$ true
    E.truelist = makelist(nextinstr)
    emit(goto —)

Boolean Expression

E $\rightarrow$ id_1 relop id_2
    E.truelist = makelist(nextinstr)
    E.falselist = makelist(nextinstr+ 1)
    emit(if id_1 relop id_2 goto —)
    emit(goto —)
E $\rightarrow$ true
    E.truelist = makelist(nextinstr)
    emit(goto —)
E $\rightarrow$ false
    E.falselist = makelist(nextinstr)
    emit(goto —)

Boolean Expression

$E \rightarrow id_1$ relop id_2
- E.truelist = makelist(nextinstr)
- E.falselist = makelist(nextinstr+ 1)
- emit(if id_1 relop id_2 goto —)
- emit(goto —)

$E \rightarrow$ true
- E.truelist = makelist(nextinstr)
- emit(goto —)

$E \rightarrow$ false
- E.falselist = makelist(nextinstr)
- emit(goto —)

$M \rightarrow \epsilon$
- M.instr = nextinstr

Boolean Expression

$E \rightarrow E_1$ or M E_2

backpatch(E_1.falselist, M.instr)

E.truelist = merge(E_1.truelist, E_2.truelist)

E.falselist = E_2.falselist

Boolean Expression

E $\rightarrow$ E_1 or M E_2

backpatch(E_1.falselist, M.instr)

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.instr)

E.truelist = E_2.truelist

E.falselist = merge(E_1.falselist, E_2.falselist)

Boolean Expression

E $\rightarrow$ E_1 or M E_2

backpatch(E_1.falselist, M.instr)

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.instr)

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

Boolean Expression

E $\rightarrow$ E_1 or M E_2

backpatch(E_1.falselist, M.instr)

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.instr)

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 $\rightarrow$ (E1)

E.truelist = E_1.truelist

E.falselist = E_1.falselist

Flow of Control Statements (using backpatching)

S $\rightarrow$ if E then S_1
| if E then S_1 else S_2
| while E do S_1
| begin L end
| A
L $\rightarrow$ L ; S
| S
S : Statement
A : Assignment
L : Statement list

Scheme to implement translation

- E has attributes `truelist` and `falselist`

Scheme to implement translation

- E has attributes `truelist` and `falselist`
- L and S have a list of unfilled `nextinstr` to be filled by backpatching

Scheme to implement translation

- E has attributes `truelist` and `falselist`
- L and S have a list of unfilled `nextinstr` to be filled by backpatching
- S $\rightarrow$ while E do S_1
  requires labels S.begin and E.true

Scheme to implement translation

- E has attributes `truelist` and `falselist`
- L and S have a list of unfilled `nextinstr` to be filled by backpatching
- S $\rightarrow$ while E do S_1
  requires labels S.begin and E.true
  - markers M_1 and M_2 record these labels

Scheme to implement translation

- E has attributes `truelist` and `falselist`
- L and S have a list of unfilled `nextinstr` to be filled by backpatching
- S $\rightarrow$ while E do S_1
  requires labels S.begin and E.true
  - markers M_1 and M_2 record these labels
  - S $\rightarrow$ while M_1 E do M_2 S_1

Scheme to implement translation

- E has attributes `truelist` and `falselist`
- L and S have a list of unfilled `nextinstr` to be filled by backpatching
- S $\rightarrow$ while E do S_1
  requires labels S.begin and E.true
  - markers M_1 and M_2 record these labels
  - S $\rightarrow$ while M_1 E do M_2 S_1
  - when `while` is reduced to S

Scheme to implement translation

- E has attributes `truelist` and `falselist`
- L and S have a list of unfilled `nextinstr` to be filled by backpatching
- S $\rightarrow$ while E do S_1
  requires labels S.begin and E.true
  - markers M_1 and M_2 record these labels
  - S $\rightarrow$ while M_1 E do M_2 S_1
  - when `while` is reduced to S
    backpatch $S_1.nextlist$ to make target of all the statements to M_1.instr

Scheme to implement translation

- E has attributes `truelist` and `falselist`
- L and S have a list of unfilled `nextinstr` to be filled by backpatching
- S $\rightarrow$ while E do S_1
  requires labels S.begin and E.true
  - markers M_1 and M_2 record these labels
  - S $\rightarrow$ while M_1 E do M_2 S_1
  - when `while` is reduced to S
    backpatch S_1.*nextlist* to make target of all the statements to M_1.instr
  - E.truelist is backpatched to go to the beginning of S_1 (M_2.instr)

Scheme for the translation

S $\rightarrow$ if E then M S_1

backpatch(E.truelist, M.instr)

S.nextlist = merge(E.falselist, S_1.nextlist)

Scheme for the translation

S $\rightarrow$ if E then M S_1
backpatch(E.truelist, M.instr)
S.nextlist = merge(E.falselist, S_1.nextlist)
S $\rightarrow$ if E them M_1 S_1 N else M_2 S_2
backpatch(E.truelist, M_1.instr)
backpatch(E.falselist, M_2.instr)
S.next = merge(S_1.nextlist, N.nextlist, S_2.nextlist)

Scheme for the translation

S $\rightarrow$ if E then M S_1
backpatch(E.truelist, M.instr)
S.nextlist = merge(E.falselist, S_1.nextlist)
S $\rightarrow$ if E them M_1 S_1 N else M_2 S_2
backpatch(E.truelist, M_1.instr)
backpatch(E.falselist, M_2.instr)
S.next = merge(S_1.nextlist, N.nextlist, S_2.nextlist)
N $\rightarrow \epsilon$ N.nextlist = makelist(nextinstr)
emit(goto —)

Scheme for the translation

S $\rightarrow$ if E then M S_1
backpatch(E.truelist, M.instr)
S.nextlist = merge(E.falselist, S_1.nextlist)
S $\rightarrow$ if E them M_1 S_1 N else M_2 S_2
backpatch(E.truelist, M_1.instr)
backpatch(E.falselist, M_2.instr)
S.next = merge(S_1.nextlist, N.nextlist, S_2.nextlist)
N $\rightarrow \epsilon$ N.nextlist = makelist(nextinstr)
emit(goto —)
S $\rightarrow$ while M_1 E do M_2 S_1
backpatch(S_1.nextlist, M_1.instr)
backpatch(E.truelist, M_2.instr)
S.nextlist = E.falselist
emit (goto M_1.instr)

Procedure Calls

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

Procedure Calls

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

- Calling sequence
  - allocate space for activation record

Procedure Calls

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

- Calling sequence
  - allocate space for activation record
  - evaluate arguments

Procedure Calls

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

- Calling sequence
  - allocate space for activation record
  - evaluate arguments
  - establish environment pointers

Procedure Calls

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

- Calling sequence
  - allocate space for activation record
  - evaluate arguments
  - establish environment pointers
  - save status and return address

Procedure Calls

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

- Calling sequence
  - allocate space for activation record
  - evaluate arguments
  - establish environment pointers
  - save status and return address
  - jump to the beginning of the procedure

IR Generation for Procedure

- Generate three address codes needed to evaluate arguments which are expressions

IR Generation for Procedure

- Generate three address codes needed to evaluate arguments which are expressions
- Generate a list of param three address statements

IR Generation for Procedure

- Generate three address codes needed to evaluate arguments which are expressions
- Generate a list of param three address statements
- Store arguments in a list

IR Generation for Procedure

- Generate three address codes needed to evaluate arguments which are expressions
- Generate a list of param three address statements
- Store arguments in a list

S → call id (Elist)
    for each item p on queue do emit('param' p)
    emit('call' id.place)

IR Generation for Procedure

- Generate three address codes needed to evaluate arguments which are expressions
- Generate a list of param three address statements
- Store arguments in a list

S → call id (Elist)
    for each item p on queue do emit('param' p)
    emit('call' id.place)
Elist → Elist , E
    append E.place to the end of queue

IR Generation for Procedure

- Generate three address codes needed to evaluate arguments which are expressions
- Generate a list of param three address statements
- Store arguments in a list

S → call id (Elist)
    for each item p on queue do emit('param' p)
    emit('call' id.place)
Elist → Elist , E
    append E.place to the end of queue
Elist → E
    initialize queue to contain E.place

note that the argument that is appearing first in the statement will be printing with param first here