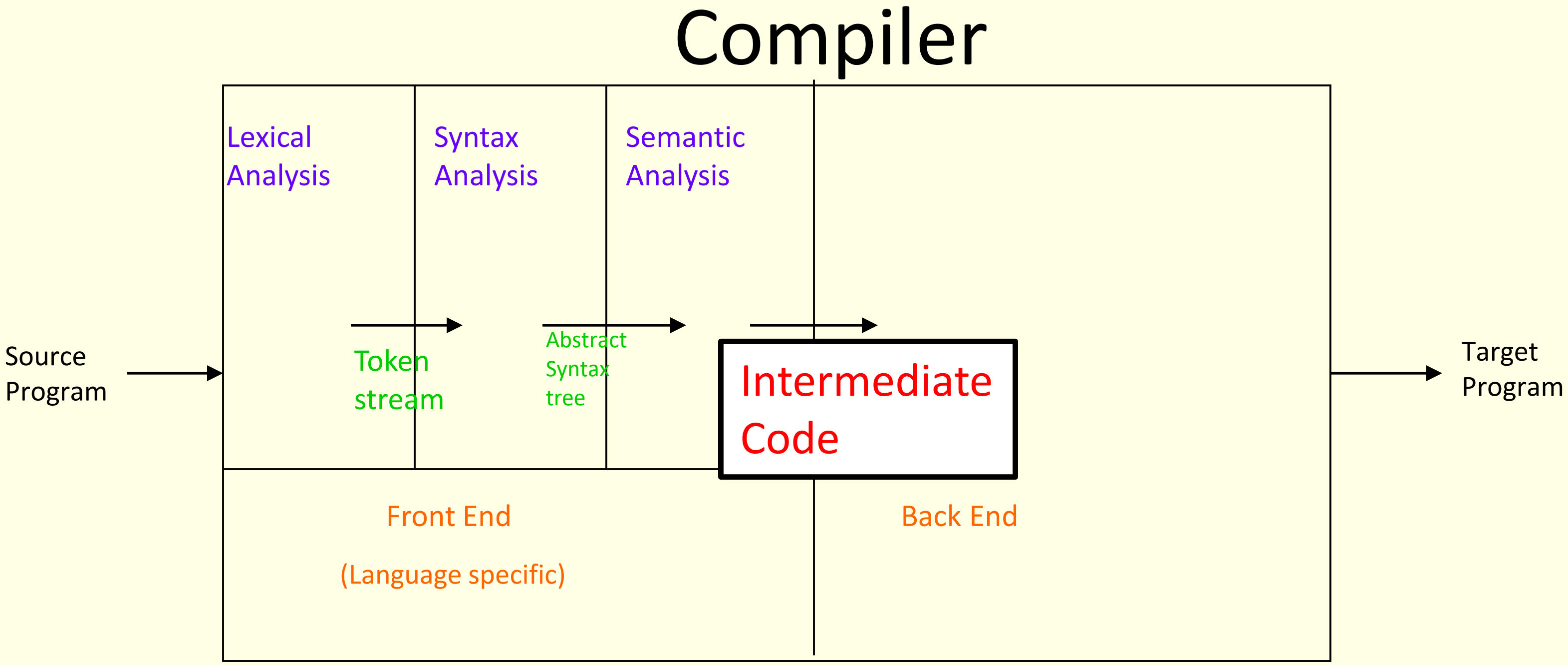
Principles of Compiler Design

Intermediate Representation



Compiler

Source Program

Lexical Analysis

Token stream

Syntax Analysis

Abstract Syntax tree

Semantic Analysis

Intermediate Code

Front End

(Language specific)

Back End

Target Program

Intermediate Code Generation

- Code generation is a mapping from source level abstractions to target machine abstractions

- Abstraction at the source level

  identifiers, operators, expressions, statements, conditionals, iteration, functions (user defined, system defined or libraries)

- Abstraction at the target level

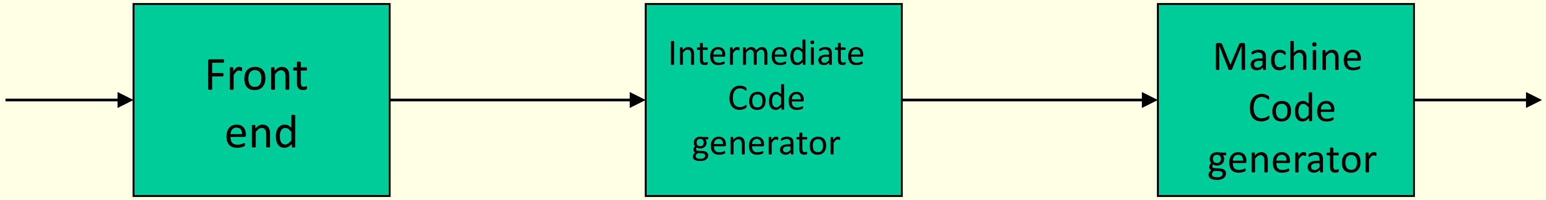
  memory locations, registers, stack, opcodes, addressing modes, system libraries, interface to the operating systems

Intermediate Code Generation ...

- Front end translates a source program into an intermediate representation

- Back end generates target code from intermediate representation

- Benefits
  - Retargeting is possible
  - Machine independent code optimization is possible



Front end

Intermediate Code generator

Machine Code generator

Three address code

- Assignment
  - x = y op z
  - x = op y
  - x = y
- Jump
  - goto L
  - if x relop y goto L
- Indexed assignment
  - x = y[i]
  - x[i] = y
- Function
  - param x
  - call p,n
  - return y
- Pointer
  - x = &y
  - x = *y
  - *x = y

Syntax directed translation of expression into 3-address code

- Two attributes
- ***E.place***, a name that will hold the value of E, and
- ***E.code***, the sequence of three-address statements evaluating E.
- A function **gen(…)** to produce sequence of three address statements
  - The statements themselves are kept in some data structure, e.g. list
  - SDD operations described using pseudo code

Syntax directed translation of expression into 3-address code

S → id := E

S.code := E.code ||

gen(id.place:= E.place)

$E \rightarrow E_1 + E_2$

E.place:= newtmp

E.code:= E_1.code || E_2.code ||

gen(E.place := E_1.place + E_2.place)

$E \rightarrow E_1 * E_2$

E.place:= newtmp

E.code := E_1.code || E_2.code ||

gen(E.place := E_1.place * E_2.place)

Syntax directed translation of expression …

$E \rightarrow -E_1$

E.place := newtmp

E.code := E_1.code ||

gen(E.place := - E_1.place)

$E \rightarrow (E_1)$

E.place := E_1.place

E.code := E_1.code

$E \rightarrow id$

E.place := id.place

E.code := ‘ ‘

Example

For a = b * -c + b * -c

following code is generated

t_1 = -c

t_2 = b * t_1

t_3 = -c

t_4 = b * t_3

t_5 = t_2 + t_4

a = t_5

Flow of Control

S → while E do S_1

Desired Translation is

S. begin :

E.code

if E.place = 0 goto S.after

S_1.code

goto S.begin

S.after :

S.begin := newlabel

S.after := newlabel

S.code := gen(S.begin:) ||
E.code ||
gen(if E.place = 0 goto S.after) ||
S_1.code ||
gen(goto S.begin) ||
gen(S.after:)

Flow of Control …

$S \rightarrow$ if E then S_1 else S_2

E.code
if E.place = 0 goto S.else
S_1.code
goto S.after
S.else:
S_2.code
S.after:

S.else := newlabel

S.after := newlabel

S.code = E.code ||
gen(if E.place = 0 goto S.else) ||
S_1.code ||
gen(goto S.after) ||
gen(S.else :) ||
S_2.code ||
gen(S.after :)

Declarations

P → D

D → D ; D

D → id : T

T → integer

T → real

Declarations

For each name create symbol table entry with information like type and relative address

P → D

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
```

Declarations

For each name create symbol table entry with information like type and relative address

P → {offset=0}   D

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

Declarations …

$T \rightarrow$ array [num] of T_1

T.type = array(num.val, T_1.type)

T.width = num.val x T_1.width

$T \rightarrow \uparrow T_1$

T.type = pointer(T_1.type)

T.width = 4

Keeping track of local information

- when a nested procedure is seen, processing of declaration in enclosing procedure is temporarily suspended

- assume following language

  P → D

  D → D ;D | id : T | proc id ;D ; S

- a new symbol table is created when procedure declaration

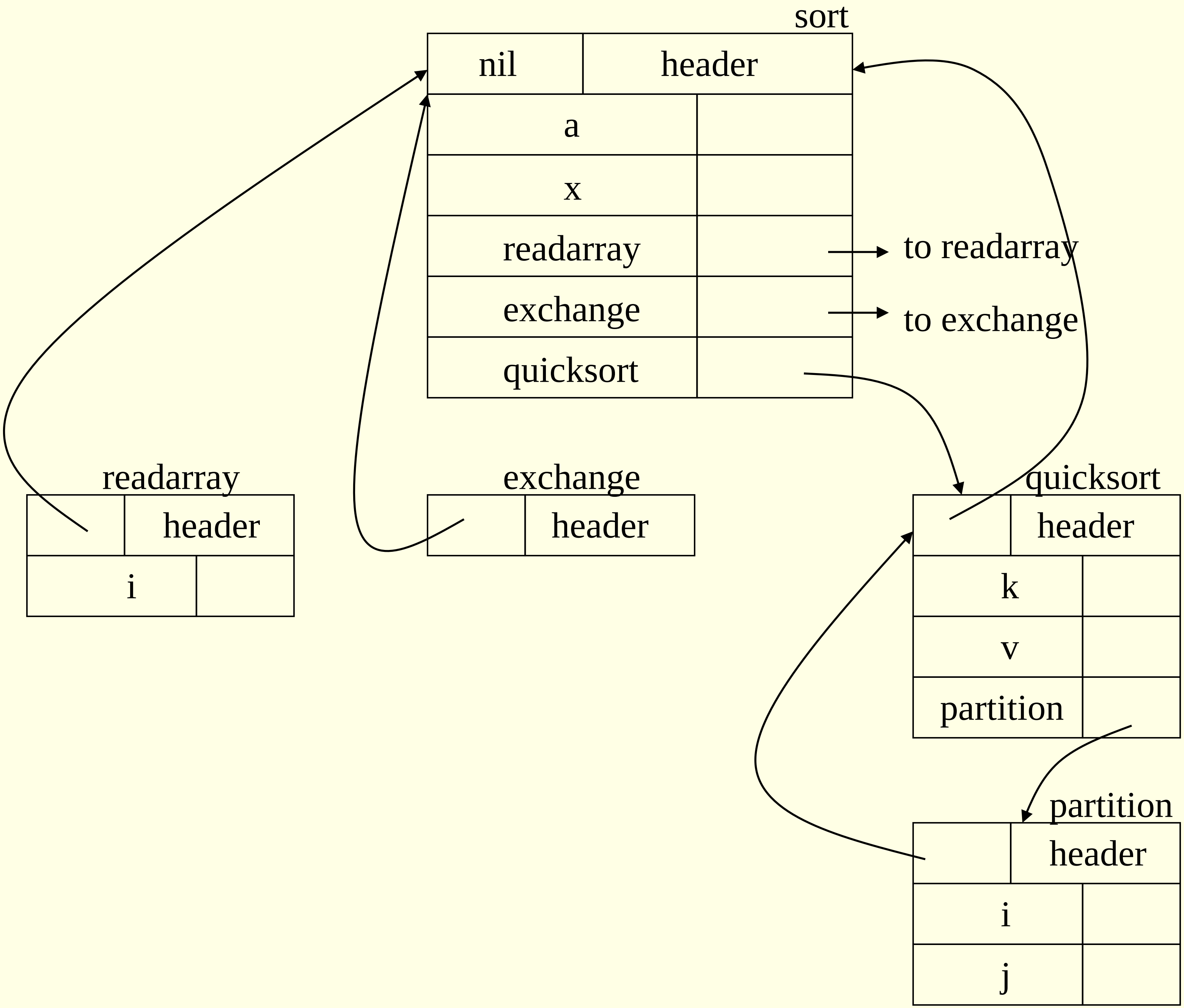
  D → proc id ; D_1 ; S   is seen

- entries for D_1 are created in the new symbol table

- the name represented by id is local to the enclosing procedure

Example

```
program sort;
    var a : array[1..n] of integer;
        x : integer;
    procedure readarray;
        var i : integer;
           ……
    procedure exchange(i,j:integers);
           ……
    procedure quicksort(m,n : integer);
        var k,v : integer;
            function partition(x,y:integer):integer;
                var i,j: integer;
                    ……
         ……
begin{main}
     ……
end.
```



sort

nil | header

a

x

readarray → to readarray

exchange → to exchange

quicksort

readarray

header

i

exchange

header

quicksort

header

k

v

partition

partition

header

i

j

Creating symbol table: Interface

- mktable (previous)

  create a new symbol table and return a pointer to the new table. The argument previous points to the enclosing procedure
- enter (table, name, type, offset)

  creates a new entry
- addwidth (table, width)

  records cumulative width of all the entries in a table
- enterproc (table, name, newtable)

  creates a new entry for procedure name. newtable points to the symbol table of the new procedure
- Maintain two stacks: (1) symbol tables and (2) offsets
- Standard stack operations: push, pop, top

Creating symbol table …

D → proc id;
{t = mktable(top(tblptr));
push(t, tblptr); push(0, offset)}
D_1; S
{t = top(tblptr);
addwidth(t, top(offset));
pop(tblptr); pop(offset);
enterproc(top(tblptr), id.name, t)}

D → id: T
{enter(top(tblptr), id.name, T.type, top(offset));
top(offset) = top (offset) + T.width}

Creating symbol table …

```
P →
            {t=mktable(nil);
            push(t,tblptr);
            push(0,offset)}
      D
            {addwidth(top(tblptr),top(offset));
            pop(tblptr); // save it somewhere!
            pop(offset)}

D →  D ; D
```

Field names in records

```
T → record
          {t = mktable(nil);
          push(t, tblptr); push(0, offset)}
    D end
          {T.type = record(top(tblptr));
          T.width = top(offset);
          pop(tblptr); pop(offset)}
```

Names in the Symbol table

```
S → id := E
        {p = lookup(id.place);
        if p <> nil then emit(p := E.place)
                    else error}

E →  id
        {p = lookup(id.name);
        if p <> nil then E.place = p
                    else error}
```

emit is like gen, but instead of returning code, it generates code as a side effect in a list of three address instructions.

Type conversion within assignments

```
$E \rightarrow E_1 + E_2$
        E.place= newtmp;
        if $E_1$.type = integer and $E_2$.type = integer
          then emit(E.place ':=' $E_1$.place 'int+' $E_2$.place);
        E.type = integer;
        …
        similar code if both $E_1$.type and $E_2$.type are real
        …
        else if $E_1$.type = int and $E_2$.type = real
          then
                u = newtmp;
                emit(u ':=' inttoreal $E_1$.place);
                emit(E.place ':=' u 'real+' $E_2$.place);
                E.type = real;
        …
        similar code if $E_1$.type is real and $E_2$.type is integer
```

Example

real x, y;

int i, j;

x = y + i * j

generates code

t_1 = i int* j

t_2 = inttoreal t_1

t_3 = y real+ t_2

x = t_3

Boolean Expressions

- compute logical values
- change the flow of control
- boolean operators are: and or not

```
E → E or E
  |  E and E
  |  not E
  |  (E)
  |  id relop id
  |  true
  |  false
```

Methods of translation

- Evaluate similar to arithmetic expressions
  - Normally use 1 for true and 0 for false

- implement by flow of control
  - given expression E_1 or E_2
    if E_1 evaluates to true
    then E_1 or E_2 evaluates to true
    without evaluating E_2

Numerical representation

- a or b and not c

  t_1 = not c
  t_2 = b and t_1
  t_3 = a or t_2

- relational expression a < b is equivalent to
  if a < b then 1 else 0

  1. if a < b goto 4.
  2. t = 0
  3. goto 5
  4. t = 1
  5.

Syntax directed translation of boolean expressions

$E \rightarrow E_1$ or E_2

E.place := newtmp
emit(E.place ':=' E_1.place 'or' E_2.place)

$E \rightarrow E_1$ and E_2

E.place:= newtmp
emit(E.place ':=' E_1.place 'and' E_2.place)

$E \rightarrow$ not E_1

E.place := newtmp
emit(E.place ':=' 'not' E_1.place)

$E \rightarrow (E_1)$ E.place = E_1.place

Syntax directed translation of boolean expressions

```
E → id1 relop id2
          E.place := newtmp
          emit(if id1.place relop id2.place goto nextstat+3)
          emit(E.place = 0)
          emit(goto nextstat+2)
          emit(E.place = 1)

E → true
          E.place := newtmp
          emit(E.place = '1')

E → false
          E.place := newtmp
          emit(E.place = '0')
```

"nextstat" is a global variable; a pointer to the statement to be emitted. emit also updates the nextstat as a side-effect.

Example:
Code for a < b or c < d and e < f

```
100: if $a < b$ goto 103
101: $t_l = 0$
102: goto 104
103: $t_l = 1$
104:
     if $c < d$ goto 107
105: $t_2 = 0$
106: goto 108
107: $t_2 = 1$
108:
     if $e < f$ goto 111
109: $t_3 = 0$
110: goto 112
111: $t_3 = 1$
112:
     $t_4 = t_2$ and $t_3$
113: $t_5 = t_l$ or $t_4$
```

Short Circuit Evaluation of boolean expressions

- Translate boolean expressions without:
  - generating code for boolean operators
  - evaluating the entire expression

- Flow of control statements

$$
\begin{aligned}
S &\rightarrow \text{if } E \text{ then } S_1 \\
&\mid \text{if } E \text{ then } S_1 \text{ else } S_2 \\
&\mid \text{while } E \text{ do } S_1
\end{aligned}
$$

Each Boolean expression E has two attributes, **true** and **false**. These attributes hold the label of the **target stmt** to jump to.

Control flow translation of boolean expression

if E is of the form: $a < b$

then code is of the form: if $a < b$ goto E.true

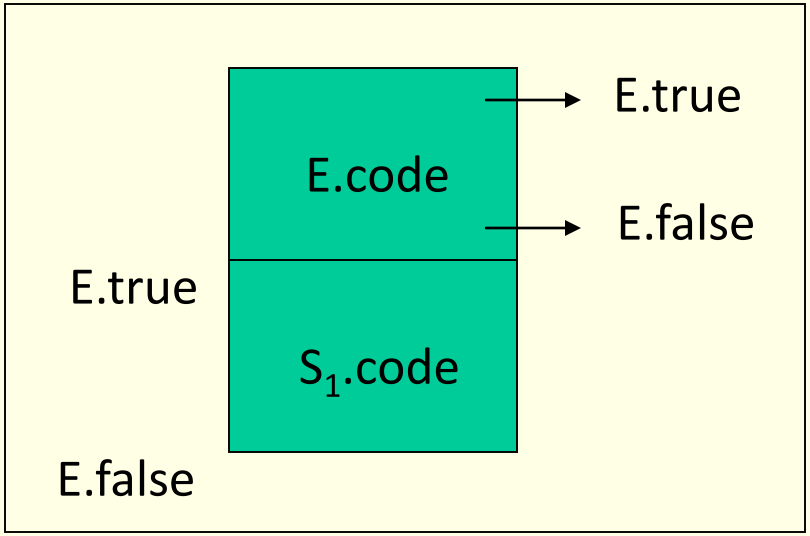
goto E.false

$E \rightarrow id_1$ relop id_2

E.code = gen(if id_1 relop id_2 goto E.true) || gen(goto E.false)

$E \rightarrow$ true E.code = gen(goto E.true)

$E \rightarrow$ false E.code = gen(goto E.false)



$S \rightarrow$ if E then S_1

E.true = newlabel

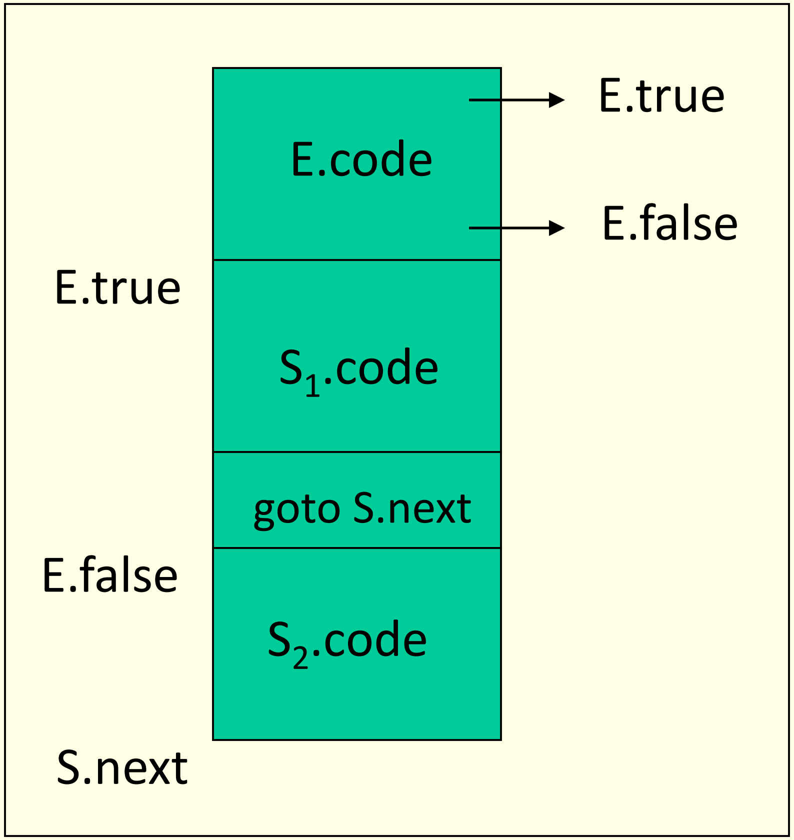
E.false = S.next

S_1.next = S.next

S.code = E.code ||

gen(E.true ':') ||

S_1.code



S → 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 ||

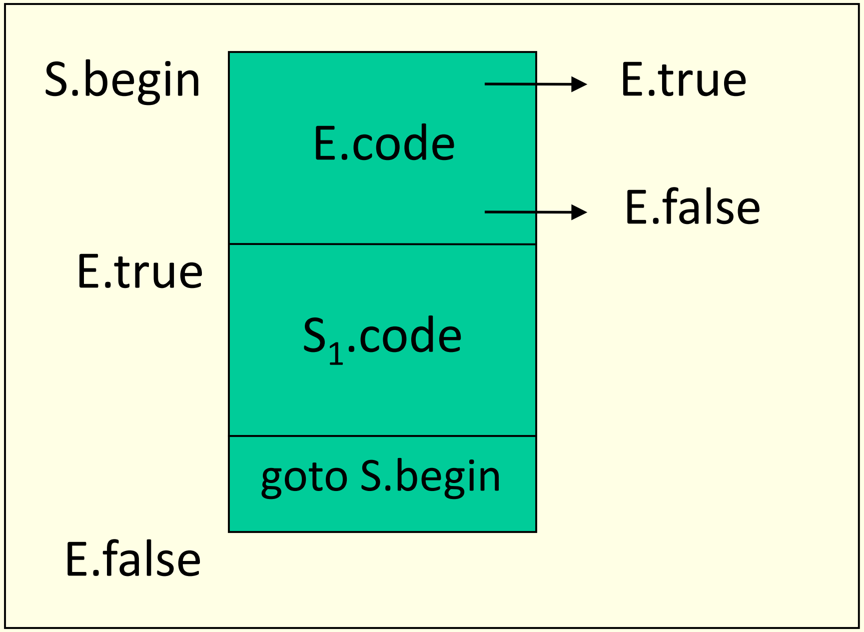
gen(E.true ':') ||

S_1.code ||

gen(goto S.next) ||

gen(E.false ':') ||

S_2.code



S.begin

E.code

E.true

E.false

E.true

S_1.code

goto S.begin

E.false

S → while E do S_1

S.begin = newlabel

E.true = newlabel

E.false = S.next

S_1.next = S.begin

S.code = gen(S.begin ':') ||
E.code ||
gen(E.true ':') ||
S_1.code ||
gen(goto S.begin)

Control flow translation of boolean expression

$E \rightarrow E_1$ or E_2

E_1.true := E.true
E_1.false := newlabel
E_2.true := E.true
E_2.false := E.false
E.code := E_1.code || gen(E_1.false) || E_2.code

$E \rightarrow E_1$ and E_2

E_1.true := newlabel
E_1 false := E.false
E_2.true := E.true
E_2 false := E.false
E.code := E_1.code || gen(E_1.true) || E_2.code

Control flow translation of boolean expression …

$E \rightarrow \text{not } E_1$

$E_1.\text{true} := E.\text{false}$

$E_1.\text{false} := E.\text{true}$

$E.\text{code} := E_1.\text{code}$

$E \rightarrow (E_1)$

$E_1.\text{true} := E.\text{true}$

$E_1.\text{false} := E.\text{false}$

$E.\text{code} := E_1.\text{code}$

Example

Code for a < b or c < d and e < f

```
        if a < b goto Ltrue
        goto L1
L1:     if c < d goto L2
        goto Lfalse
L2:     if e < f goto Ltrue
        goto Lfalse

Ltrue:
Lfalse:
```

Example …

Code for

```
while a < b do
      if c<d then x=y+z
         else     x=y-z
```

```
L1:     if a < b goto L2
        goto Lnext
L2:     if c < d goto L3
        goto L4
L3:     $t_1$ = Y + Z
        X= $t_1$
        goto L1
L4:     $t_1$ = Y - Z
        X= $t_1$
        goto L1
Lnext:
```

Case Statement

-
  ```
  switch expression
    begin
          case value: statement
          case value: statement
          ….
          case value: statement
          default: statement
    end
  ```

- evaluate the expression
- find which value in the list of cases is the same as the value of the expression.
  - Default value matches the expression if none of the values explicitly mentioned in the cases matches the expression
- execute the statement associated with the value found

Translation

```
            code to evaluate E into t
            if t <> V1 goto L1
            code for S1
            goto next
L1          if t <> V2 goto L2
            code for S2
            goto next
L2:      ……
Ln-2        if t <> Vn-l goto Ln-l
            code for Sn-l
            goto next
Ln-1:       code for Sn
next:
```

```
            code to evaluate E into t
            goto test
L1: code for S1
            goto next
L2: code for S2
            goto next
            ……
Ln: code for Sn
            goto next
test:       if t = V1 goto L1
            if t = V2 goto L2
            ….
            if t = Vn-1 goto Ln-1
            goto Ln
next:
```

Efficient for n-way branch

BackPatching

- way to implement boolean expressions and flow of control statements in one pass
- code is generated as quadruples into an array
- 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 Expressions

$E \rightarrow E_1 \text{ or } M\ E_2$
$\quad \mid E_1 \text{ and } M\ E_2$
$\quad \mid \text{not } E_1$
$\quad \mid (E_1)$
$\quad \mid id_1 \text{ relop } id_2$
$\quad \mid \text{true}$
$\quad \mid \text{false}$

$M \rightarrow \epsilon$

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

- 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.quad records the number of the first statement of E_2.code

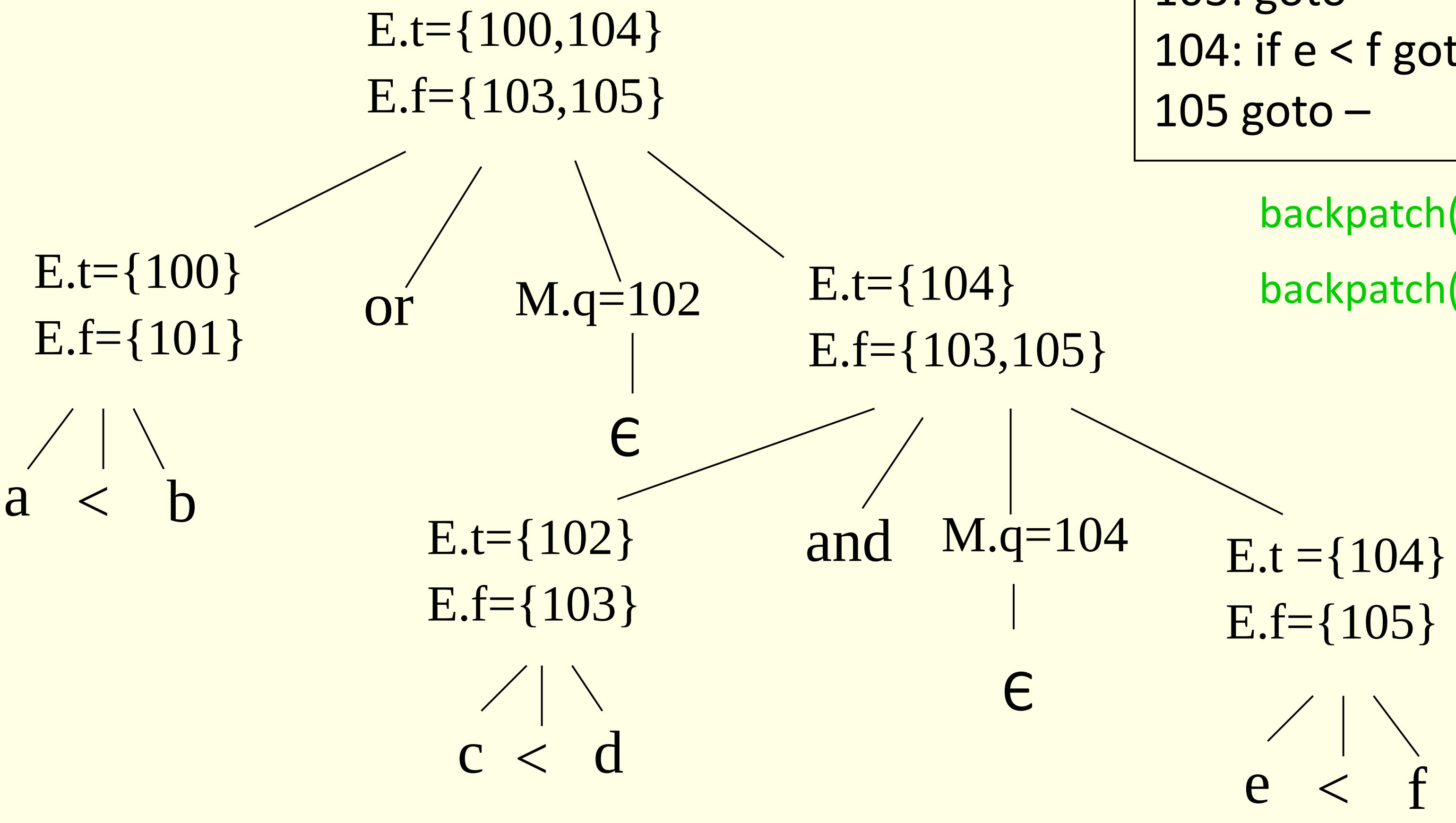
$E \rightarrow E_1$ or M E_2
backpatch(E_1.falselist, M.quad)
E.truelist = merge(E_1.truelist, E_2.truelist)
E.falselist = E_2.falselist
$E \rightarrow E_1$ and M E_2
backpatch(E_1.truelist, M.quad)
E.truelist = E_2.truelist
E.falselist = merge(E_1.falselist, E_2.falselist)
$E \rightarrow$ not E_1
E.truelist = E_1 falselist
E.falselist = E_1.truelist
$E \rightarrow (E_1)$
E.truelist = E_1.truelist
E.falselist = E_1.falselist

```
E → $id_1$ relop $id_2$
        E.truelist = makelist(nextquad)
        E.falselist = makelist(nextquad+ 1)
        emit(if $id_1$ relop $id_2$ goto --- )
        emit(goto ---)
E → true
        E.truelist = makelist(nextquad)
        emit(goto ---)
E → false
        E.falselist = makelist(nextquad)
        emit(goto ---)
M →  Є
        M.quad = nextquad
```

Generate code for
a < b or c < d and e < f

Initialize nextquad to 100

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



backpatch(102,104)

backpatch(101,102)

Flow of Control Statements

$S \rightarrow$ 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
- L and S have a list of unfilled quadruples to be filled by backpatching
- S → while E do S_1

  requires labels S.begin and E.true
  - markers M_1 and M_2 record these labels

    S → 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.quad
  - E.truelist is backpatched to go to the beginning of S_1 (M_2.quad)

Scheme to implement translation …

$S \rightarrow$ if E then M S_1

backpatch(E.truelist, M.quad)

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.quad)

backpatch(E.falselist, M_2.quad)

S.next = merge(S_1.nextlist,

N.nextlist,

S_2.nextlist)

Scheme to implement translation …

$S \rightarrow$ 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 M_1.quad)

Scheme to implement translation …

$S \rightarrow$ begin L end	S.nextlist = L.nextlist
$S \rightarrow A$	S.nextlist = makelist()
$L \rightarrow L_1$; M S	backpatch(L_1.nextlist, M.quad) L.nextlist = S.nextlist
$L \rightarrow S$	L.nextlist = S.nextlist
$N \rightarrow \in$	N.nextlist = makelist(nextquad) emit(goto ---)
$M \rightarrow \in$	M.quad = nextquad