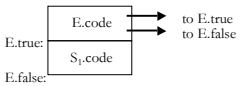
Intermediate Code Generation

Control-Flow Statements
Short-Circuit Predicate Evaluation
Back-patching

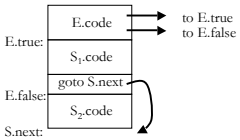
Control Flow Statements: Code Layout

- Attributes:
  - E.true: the label to which control flows if E is true
  - E.false: the label to which control flows if E is false
  - S.next: an inherited attribute with the symbolic label of the code following S

$S \rightarrow$ if E then S_1

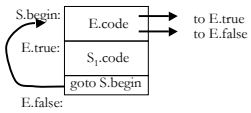


$S \rightarrow$ if E then S_1 else S_2



Code Layout

$S \rightarrow$ while E do S_1

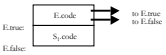


- Difficulty: Need to know where to jump to
  - Introduce a symbolic labels using the `newlabel` function
  - Use inherited attributes
  - Backpatch it later with the actual value (later…)

Grammar and Actions

`S → if E then S1 ‖`

```
E.true = newlabel
E.false = S.next
S1.next = S.next
S.code = append(E.code,gen(E.true:),S1.code)
```

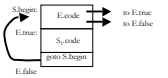


`S → if E then S1 else S2 ‖`

```
E.true = newlabel
E.false = newlabel
S1.next = S.next
S2.next = S.next
S.code = append(E.code,gen(E.true:),S1.code,
          gen(goto S.next),gen(E.false :),S2.code)
```

`S → while E do S1 ‖`

```
S.begin = newlabel
E.true = newlabel
E.false = S.next
S1.next = S.begin
S.code = append(gen(S.begin:),E.code,gen(E.true:),S1.code,
                gen(goto S.begin)
```



Control Flow Translation of Boolean Expressions

- Short-Circuit Evaluation
  - No Need to Evaluate portions of the expression if the outcome is already determined
  - Examples:
    - E_1 **or** E_2 need not evaluate E_2 if E_1 is known to be true.
    - E_1 **and** E_2 need not evaluate E_2 if E_1 is known to be false.
- Use Control Flow
  - Jump over code that evaluates boolean terms of the expression
  - Use Inherited E.false and E.true attributes and link evaluation of E

Control Flow for Boolean Expressions

$E \rightarrow id_1$ relop id_2 ‖
E.code = append(gen(if id_1.place relop id_2.place 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)

$E \rightarrow E_1$ or E_2 ‖
E_1.true = E.true
E_1.false = newlabel
E_2.true = E.true
E_2.false = E.false
E.code = append(E_1.code,gen(E_1.false:),E_2.code)

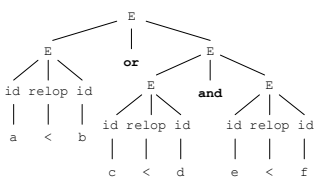
$E \rightarrow E_1$ and E_2 ‖
E_1.false = E.false
E_1.true = newlabel
E_2.true = E.true
E_2.false = E.false
E.code = append(E_1.code,gen(E_1.true:),E_2.code)

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

$E \rightarrow$ (E_1) ‖
E_1.true = E.true
E_1.false = E.false
E.code = E_1.code

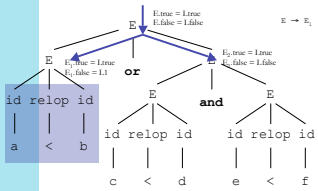
Short Circuit Evaluation

a < b **or** c < d **and** e < f



Short Circuit Evaluation

a < b **or** c < d **and** e < f



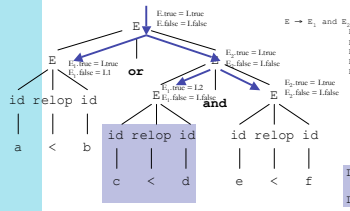
$E \rightarrow id_1$ relop id_2 ‖
E.code = append(
gen(if id_1.place relop id_2.place goto E.true),
gen(goto E.false))

$E \rightarrow E_1$ or E_2 ‖
E_1.true = E.true
E_1.false = newlabel
E_2.true = E.true
E_2.false = E.false
E.code = append(E_1.code,
gen(E_1.false:),E_2.code)

```
    if a < b goto Ltrue
    goto L1
L1:
```

Short Circuit Evaluation

a < b **or** c < d **and** e < f



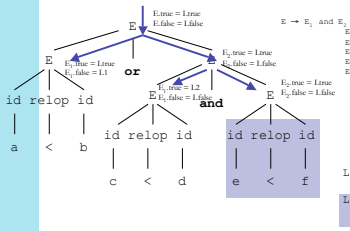
E → id$_1$ relop id$_2$ ‖
E.code = append(
gen(if id$_1$.place relop id$_2$.place goto E.true),
gen(goto E.false))

E → E$_1$ and E$_2$ ‖
E$_1$.false = E.false
E$_1$.true = newlabel
E$_2$.true = E.true
E$_2$.false = E.false
E.code = append(E$_1$.code,
gen(E$_1$.true:),E$_2$.code)

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

Short Circuit Evaluation

a < b **or** c < d **and** e < f



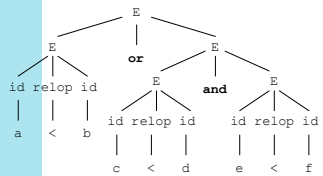
E → id$_1$ relop id$_2$ ‖
E.code = append(
gen(if id$_1$.place relop id$_2$.place goto E.true),
gen(goto E.false))

E → E$_1$ and E$_2$ ‖
E$_1$.false = E.false
E$_1$.true = newlabel
E$_2$.true = E.true
E$_2$.false = E.false
E.code = append(E$_1$.code,
gen(E$_1$.true:),E$_2$.code)

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

Short Circuit Evaluation

a < b **or** c < d **and** e < f



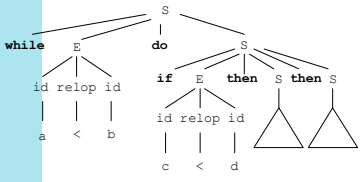

```
    if a < b goto Ltrue
    goto L1
L1: if c < d goto L2
    goto Lfalse
L2: if e < f goto Ltrue
    goto Lfalse
```

Combining Boolean and Control Flow Statements

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

S → while E do S$_1$ ‖

S.begin = newlabel
E.true = newlabel
E.false = S.next
S$_1$.next = S.begin
S.code = append(gen(S.begin:),E.code,
gen(E.true:),S$_1$.code,
gen(goto S.begin)

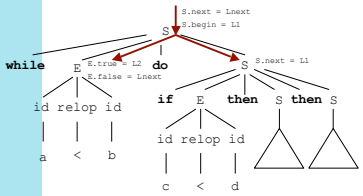


Combining Boolean and Control Flow Statements

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

$S \rightarrow$ while E do S_1 ‖

```
S.begin = newlabel
E.true = newlabel
E.false = S.next
S1.next = S.begin
S.code = append(gen(S.begin:),E.code,
                gen(E.true:),S1.code,
                gen(goto S.begin))
```

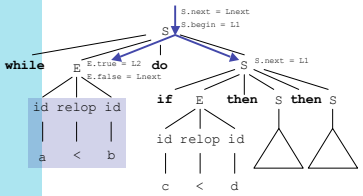


Combining Boolean and Control Flow Statements

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

$S \rightarrow$ while E do S_1 ‖

```
S.begin = newlabel
E.true = newlabel
E.false = S.next
S1.next = S.begin
S.code = append(gen(S.begin:),E.code,
                gen(E.true:),S1.code,
                gen(goto S.begin))
```



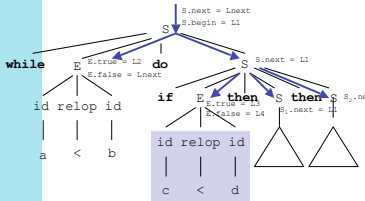

```
L1: if a < b goto L2
    goto Lnext
L2:
```

Combining Boolean and Control Flow Statements

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

$S \rightarrow$ while E do S_1 ‖

```
S.begin = newlabel
E.true = newlabel
E.false = S.next
S1.next = S.begin
S.code = append(gen(S.begin:),E.code,
                gen(E.true:),S1.code,
                gen(goto S.begin))
```



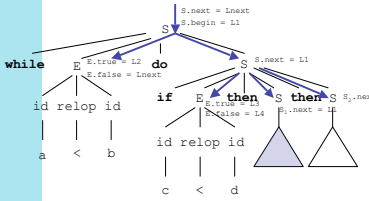

```
L1: if a < b goto L2
    goto Lnext
L2: if c < d goto L3
    goto L4
L3:
```

Combining Boolean and Control Flow Statements

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

$S \rightarrow$ while E do S_1 ‖

```
S.begin = newlabel
E.true = newlabel
E.false = S.next
S1.next = S.begin
S.code = append(gen(S.begin:),E.code,
                gen(E.true:),S1.code,
                gen(goto S.begin))
```



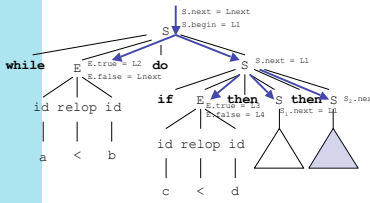

```
L1: if a < b goto L2
    goto Lnext
L2: if c < d goto L3
    goto L4
L3: t1 = x + z
    x = t1
    goto L1
L4:
```

Combining Boolean and Control Flow Statements

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

$S \rightarrow$ while E do S_1 ‖

```
S.begin = newlabel
E.true = newlabel
E.false = S.next
S1.next = S.begin
S.code = append(gen(S.begin:),E.code,
                gen(E.true:),S1.code,
                gen(goto S.begin))
```



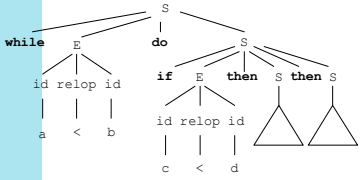

```
L1: if a < b goto L2
    goto Lnext
L2: if c < d goto L3
    goto L4
L3: t1 = x + z
    x = t1
    goto L1
L4: t2 = x - z
    x = t2
    goto L1
Lnext:
```

Combining Boolean and Control Flow Statements

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

$S \rightarrow$ while E do S_1 ‖

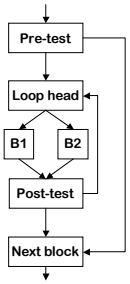
```
S.begin = newlabel
E.true = newlabel
E.false = S.next
S1.next = S.begin
S.code = append(gen(S.begin:),E.code,
                gen(E.true:),S1.code,
                gen(goto S.begin))
```




```
L1: if a < b goto L2
    goto Lnext
L2: if c < d goto L3
    goto L4
L3: t1 = x + z
    x = t1
    goto L1
L4: t2 = x - z
    x = t2
    goto L1
Lnext:
```

Loop Constructs

- **Loops**
  - Evaluate condition before loop (if needed)
  - Evaluate condition after loop
  - Branch back to the top (if needed)
- **Why this structure?**
  - Merges test with last block of loop body
  - Pre-test block to hold loop-invariant code
  - Post-test for increment instructions and test
- while, for, do, & until all fit this basic model



Break & Skip Statements

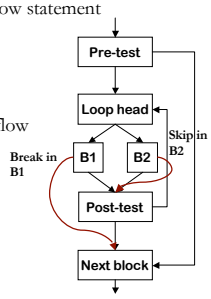
Many modern programming languages include a `break`

- Exits from the innermost control-flow statement
  - Out of the innermost loop
  - Out of a case statement

Translates into a jump

- Targets statement outside control- flow construct
- Creates multiple-exit construct
- `skip` in loop goes to next iteration

Only make sense if loop has > 1 block



Break and Skip Statements

- Need to Keep track of enclosing control-flow constructs
- Harder to have clean SDT scheme…
  - Keep a Stack of control-flow constructs
  - Using S.next as in the stack as the target for the break statement
  - For skip statements need to keep track of the label of the code of the post-test block to advance to the next iteration. This is harder since the code has not been generated yet.
- Backpatching helps
  - Use a breaklist and a skiplist to be patched later.

Case Statements

`case` or `switch` Statements Semantics

1 Evaluate the controlling expression
2 Branch to the selected case
3 Execute the code for that case
4 Branch to the statement after the case

Parts 1, 3, & 4 are well understood, part 2 is the key

- Strategies
  - Linear search (nested if-then-else constructs)
  - Build a table of case expressions & binary search it
  - Directly compute an address (requires dense case set)

Surprisingly many compilers do this for all cases!

Case Statement: Code Layout

```
switch E
 begin
  case V1: S1
  case V2: S2
  …
  case Vn-1: Sn-1
  default: Sn
 end
```

```
        code to Evaluate E into t
        goto Ltest
L1:     code for S1
        goto Lnext
L2:     code for S2
        goto Lnext
        …
Ln-1:   code for Sn-1
        goto Lnext
Ln:     code for Sn
        goto Lnext
Ltest:  if t = V1 goto L1
        if t = V2 goto L2
        …
        if t = Vn-1 goto Ln-1
        goto Ln
Lnext:
```

Linear Search

SDT scheme for Case Statements

- Issue: Need to Save the Labels and Values for the various cases for the test code at the end
  - Use a Right-Recursive Grammar
    - Use queue to save pairs (value,label) for generation of search code
    - In the end pop values from queue to generate the linear search
  - Use a Left-Recursive Grammar
    - Cleaner; No queue is needed
    - Use of the parsing stack to accumulate the non-terminals and corresponding attribute

Grammar and Actions

```
S → switch E List end  ‖
                S.code = append(E.code,gen('goto Ltest'),List.code,gen('Ltest:')
                while(queue not empty) do {
                  (vi,Li) = pop.queue;
                  if (vi = default)
                    S.code = append(S.code,gen('goto Li'));
                  else
                    S.code = append(S.code,gen('if t = vi goto Li'));


Case → case Value : S  ‖
                    Case.code = append(gen('Li:'),S.code,gen('goto Lnext');
                    queue.push((Value.val,Li));

List → Case ; List1   ‖
                    List.code = append(Case.code,List1.code);

List → default : S    ‖
                    List.code = append(gen('Li:'),S.code,gen('goto Lnext');
                    queue.push((default,Li))

List →       ε        ‖ List.code = append(gen('Li:'),gen('goto Lnext');
                    queue.push((default,Li))
```

Other Statements

- Declarations
  - Just save information in Symbol Table
  - For Structures, Unions compute offsets for each field
- Function Calls
  - Generate code to evaluate each argument in order into temporaries
  - Emit the call instruction using the temporary variables
- Structures, Variants Records and Unions
  - Use the offsets from symbol table for address generation
- Unstructured Control-Flow
  - Breaks the SDT scheme, just use a global table and backpatch it.

Back-patching

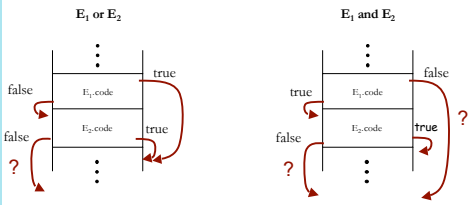
- Single Pass Solution to Code Generation?
  - No more symbolic labels - symbolic addresses instead
  - Emit code directly into an array of instructions
  - Actions associated with Productions
  - Executed when Bottom-Up Parser "Reduces" a production
- Problem
  - Need to know the labels for target branches before actually generating the code for them.
- Solution
  - Leave Branches undefined and **patch** them later
  - Requires: carrying around a list of the places that need to be patched until the value to be patched with is known.

Boolean Expressions Revisited

- Use Additional ε-Production
  - Just a Marker M
  - Label Value M.addr
- Attributes:
  - E.truelist: code places that need to be filled-in corresponding to the evaluation of E as "true".
  - E.falselist: same for "false"

(1) $E \rightarrow E_1$ **or** $M\ E_2$
(2) | E_1 **and** $M\ E_2$
(3) | **not** E_1
(4) | $(\ E_1\ )$
(5) | id_1 relop id_2
(6) | true
(7) | false
(8) $M \rightarrow \varepsilon$

Boolean Expressions: Code Outline



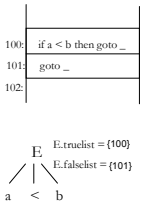
Auxiliary Functions

- Functions:
  - makelist(i): make a list with the label i
  - merge(p1,p2): creates a new list of labels with lists p1 and p2
  - backpatch(p,i): fills the locations in p with the address i
  - newAddr() : returns a new symbolic address in sequence and increments the value for the next call
- Array of Instructions
  - Linearly sequence of instructions
  - Function `emit` to generate actual instructions in the array
  - Symbolic Addresses

Using the Actions & List Attributes

(5) E → id_1 relop id_2

```
{
 E.truelist := makelist(newlabel);
 E.falselist := makelist(newlabel);
 emit("if id1.place relop id2.place goto _");
 emit("goto _");
}
```



More Actions

(3) E → not E_1	{ E.truelist := E_1.falselist; E.falselist := E_1.truelist; }
(4) E → (E_1)	{ E.truelist := E_1.truelist; E.falselist := E_1.falselist; }
(5) E → id_1 relop id_2	{ E.truelist := makelist(nextAddr()); E.falselist := makelist(nextAddr()); emit("if id_1.place relop.op id_2.place goto _"); emit("goto _"); }
(6) E → true	{ E.truelist := makelist(nextAddr()); emit("goto _"); }
(7) E → false	{ E.falselist := makelist(nextAddr()); emit("goto _"); }

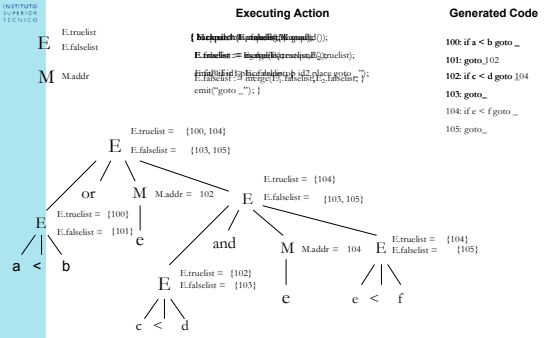
Actions

(1) E → E_1 or M E_2 { backpatch(E_1.falselist,M.Addr);
E.truelist := merge(E_1.truelist,E_2.truelist);
E.falselist := E_2.falselist; }

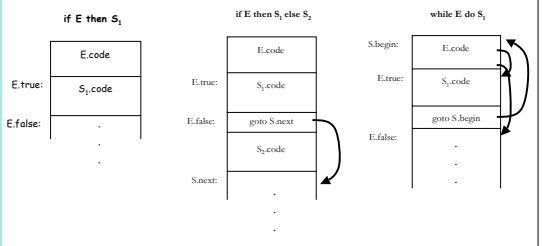
(2) E → E_1 and M E_2 { backpatch(E_1.truelist,M.Addr);
E.truelist := E_2.truelist;
E.falselist := merge(E_1.falselist, E_2.falselist); }

(8) M → ε { M.Addr := nextAddr; }

Backpatching Example



Control Flow Code Structures



Control Flow Constructs: Conditionals

- Add the `nextlist` attribute to S and N
  - denotes the set of locations in the S code to be patched with the address that follows the execution of S
  - Can be either due to control flow or fall-through

(1) S → if E then M_1 S_1 N else M_2 S_2 { backpatch(E.truelist, M_1.addr);
backpatch(E.falselist,M_2.addr);
S.nextlist := merge(S_1.nextlist,merge(N.nextlist,S_2.nextlist));
}

(2) N → ε { N.nextlist := makelist(nextAddr();
emit("goto _"); }

(3) M → ε { M.quad := nextAddr; }

(4) S → if E then M S_1 { backpatch(E.truelist, M.addr);
S.nextlist := merge(E.falselist,S_1.nextlist);
}

Control Flow Constructs: Loops

(5) $S \rightarrow$ while M_1 E do M_2 S_1 { backpatch(S_1.nextlist, M_1.addr);
backpatch(E.truelist,M_2.addr);
S.nextlist := E.falselist;
emit("goto M_1.addr");
}

Sequencing: List of Statements

- Additional Symbols
  - L for list of statements
  - S for Assignment statement

(6) $S \rightarrow$ begin L end { S.nextlist = L.nextlist; }

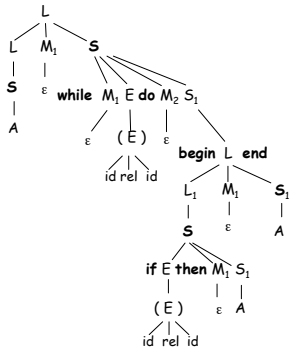
(7) $S \rightarrow A$ { S.nextlist = nil; }

(8) $L \rightarrow L_1$; M S { backpatch(L_1.nextlist, M.addr); L.nextlist = S.nextlist; }

(9) $L \rightarrow S$ { L.nextlist = S.nextlist; }

Extended Example

```
i = 0;
while (i < n) do
 begin
  if(a <= k) then
   a = a + 1;
  i++;
 end
```



Summary

- Intermediate Code Generation
  - Using Syntax-Directed Translation Schemes
  - Conditional
  - Boolean using Short-Circuit Evaluation
  - Control-Flow
- Back-patching
  - Allows Code Generation in a Single Pass