Assignment Statements

Assignment Statements

P → M D
M → ε
D → D ; D | **id** : T | proc **id** N D ; S
N → ε

S → *S* **;** *S*
S → **id :=** *E*

```
{ p := lookup(id.name);
  if p = nil then
     error()
  else
     emit(id.place ':=' E.place)
}
```

Assignment Statements Cont…

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

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

$E \rightarrow$ **id** { p := *lookup*(**id**.name);
if p = nil **then** *error*()
else
E.place := **p**
}

Type conversions within assignments

- Reject certain mixed type conversions

  Or

- Generate appropriate type conversion
- Consider only two datatype integer and real
- Consider the grammar for assignment statement.
- Introduce a new attribute E.Type

Semantic action for $E \rightarrow E_1 + E_2$

```
E.place := newtemp();
if E1.Type:=integer and E2.Type:= integer then
{   emit(E.place ':=' E1.place 'int+' E2.place);
    E.Type:=integer;          }
else if E1.Type:=real and E2.Type:= real then
{   emit(E.place ':=' E1.place 'real+' E2.place);
    E.Type:=real;  }
else if E1.Type:=integer and E2.Type:= real then
{   u:= newtemp();
    emit(u:='inttoreal' E1.place);
    emit(E.place ':=' u 'real+' E2.place);
    E.Type:=real;  }
else if E1.Type:=real and E2.Type:= integer then
{   u:= newtemp();
    emit(u:='inttoreal' E2.place);
    emit(E.place ':=' E1.place 'real+' u);
    E.Type:=integer;          }
else E.Type:=error
```

real x,y;

int i,j;

x:=y+i*j;

TAC:

t1:=I int * j

t2=inttoreal t1

t3:=y real t2

x:=t3

Boolean Expressions

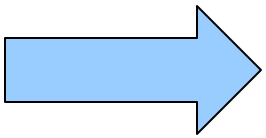
Boolean Expressions

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

1. Numerical Representation
2. Flow of control Statements

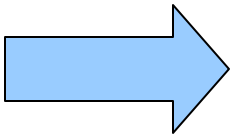
Numerical Representation

a or b and not c



```
t1 := not c
t2 := b and t1
t3 := a or t2
```

a < b



```
100: if a < b goto 103
101: t1 := 0
102: goto 104
103: t1 := 1
104:
```

Translation Scheme

E → E1 *or* E2 { E.place := newtemp();
emit (E.place ‘: =’ E1.place ‘*or*’ E2.place); }

E → E1 *and* E2 { E.place := newtemp();
emit (E.place ‘: =’ E1.place ‘*and*’ E2.place);}

E → *not* E { E.place := newtemp();
emit (E.place ‘: =’ ‘*not*’ E.place);}

E → (E1) { E.place := E1.place; }

E → *id1 relop id2*{ E.place ‘: =’ newtemp();
emit (‘if’ *id1*.place *relop*.op *id2*.place , ‘*goto*’ nextstat+3);
emit (E.place ‘: =’ ‘0’);
emit (‘*goto*’ nextstat+2);
emit (E.place ‘: =’ ‘1’);}

E → *true* { E.place := newtemp();
emit (E.place ‘:=’ ‘1’);}

E → *false*{ E.place = newtemp();
emit (E.place ‘:=’ ‘0’); }

Example

Translation of a<b or c<d and e<f:

1. 100: if a<b goto 103 E.place=t1;
   101: t1 := 0
   102: goto 104
   103: t1 := 1
2. 104: if c<d goto 107 E.place=t2;
   105: t2 := 0
   106: goto 108
   107: t2 := 1
3. 108: if e<f goto 111 E.place=t3;
   109: t3 := 0
   110: goto 112
   111: t3 := 1
4. 112: t4 := t2 and t3 E.place=t4;
5. 113: t5 := t1 or t4 E.place=t5;

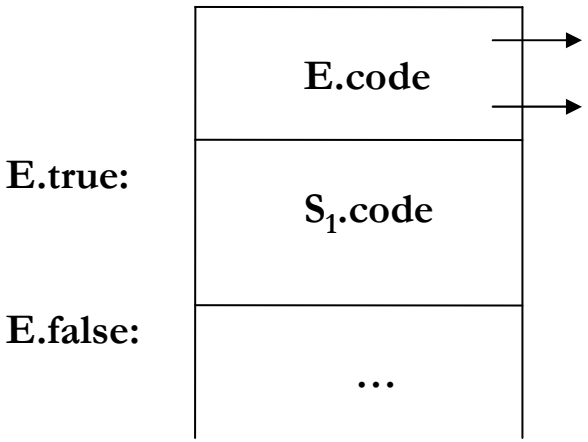
Flow of Control Statements

S → if E then S1

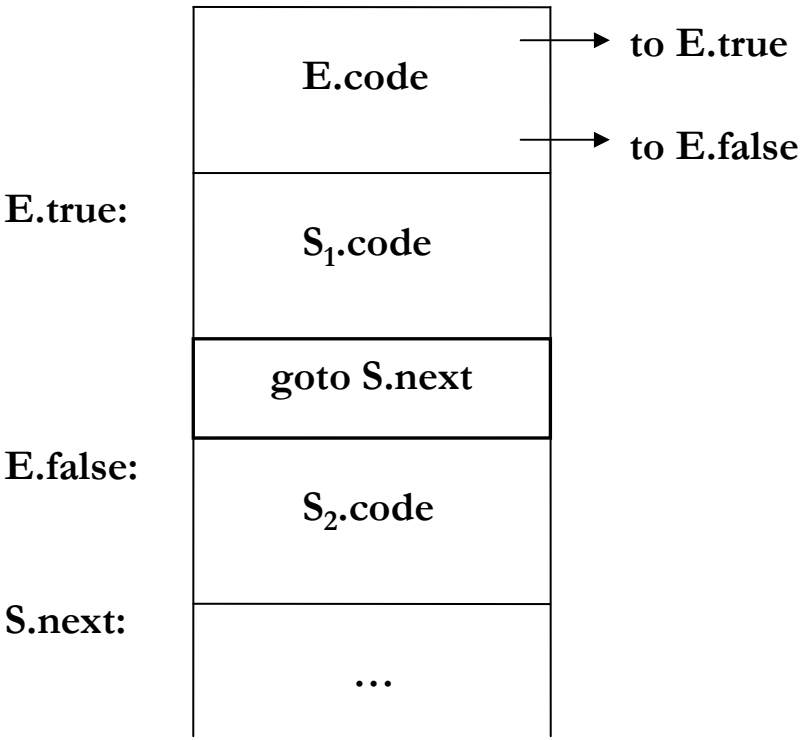
S → if E then S1 else S2

S → while E do S1

Pictorial Representation

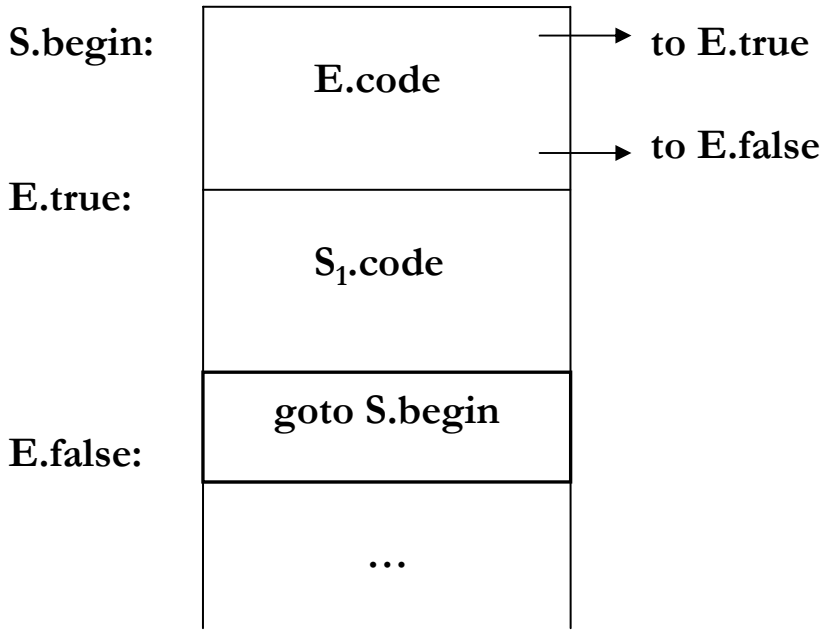


if - then



if – then - else

Pictorial Representation



while - do

Syntax directed Definition for Flow of Control Statements

S → if E then S1

```
{       E.true := newlabel ;
        E.false := S.next ;
        S1.next := S.next ;
        S.code := E.code || 
        gen(E.true ':') || S1.code
}
```

Syntax directed Definition for Flow of Control Statements

S → if E then S1 else S2

{

E.true := newlabel ;

E.false := newlabel ;

S1.next := S.next ;

S2.next := S.next ;

S.code := E.code || gen(E.true ':') ||
S1.code || gen('goto' S.next) ||
gen(E.false ':') || S2.code

}

Syntax directed Definition for Flow of Control Statements

S → while E do S1

{

S.begin := newlabel ;

E.true := newlabel ;

E.false := S.next ;

S1.next := S.begin ;

S.code := gen(S.begin ':') || E.code ||
gen(E.true ':') || S1.code || gen('goto' S.begin)

}

Control Flow translation of Boolean Expression

E → E1 *or* E2	E1.True:=E.True; E1.False:=*newlabel();* E2.True:=E.True; E2.False:=E.False; E.Code:=E1.Code ||*gen*('E1.False:') || E2.Code
E → E1 *and* E2	E1.True:= *newlabel();* E1.False:=E.False; E2.True:=E.True; E2.False:=E.False; E.Code:=E1.Code ||*gen*('E1.True:') || E2.Code

Control Flow translation of Boolean Expression

E → *not* E1	E1.True:= E.False; E1.False:=E.True; E.Code:=E1.Code
E → (E1)	E1.True:= E.True; E1.False:=E.False; E.Code:=E1.Code
E → *id1 relop id2*	E.Code:= gen('*if*' id1.place relop.op id2.place '*goto*' E.True) || *gen('goto'* E.False)
E → *true*	E.Code:= *gen('goto'* E.True)
E → *false*	E.Code:= *gen('goto'* E.False);

Case Statement

Switch Statement Syntax

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

Translation of a case statement

```
code to evaluate E into
    t
goto test
L1:     code for S1
        goto next;
L2:     code for S2
        goto next;
…
Ln-1:   code for Sn-1
        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:
```

Another Translation of a case statemen

```
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:if t≠V3 goto L3
    code for S3
    goto next
…
Ln-1:          code for Sn
Next:
```