- Source Language Instructions to IC instructions: Since IC is lower-level than SL, several IC instructions are needed to translate a single SL instruction.

  The main idea is to preserve meaning in translation. The set of IC instructions *for* a SL instruction must have the same meaning (must do the same thing computationally).

- Since we assume 1-1 correspondence of form and meaning in SL (after all, every syntactic construction is supposed to lend itself to one computation), we can define this correspodence in the form of a *template*.

  A template is a sequence of skeletal IC instructions. Details in the

template to be filled in by SL ’contents’

In other words, templates are semantic in nature; they reflect the IC counterparts of SL instructions. But since instructions make reference to data and variables, these are missing from the template.

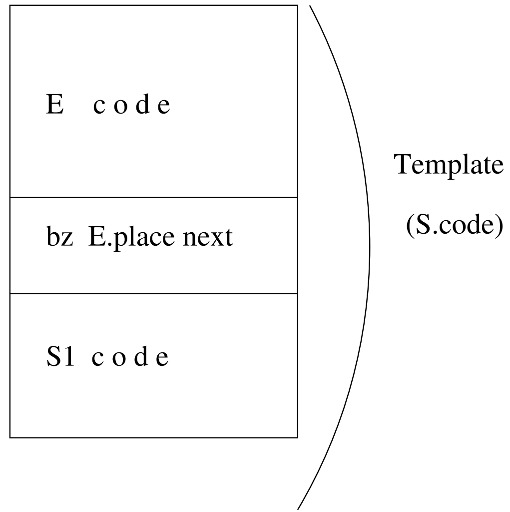
- IC templates for

  - assignment

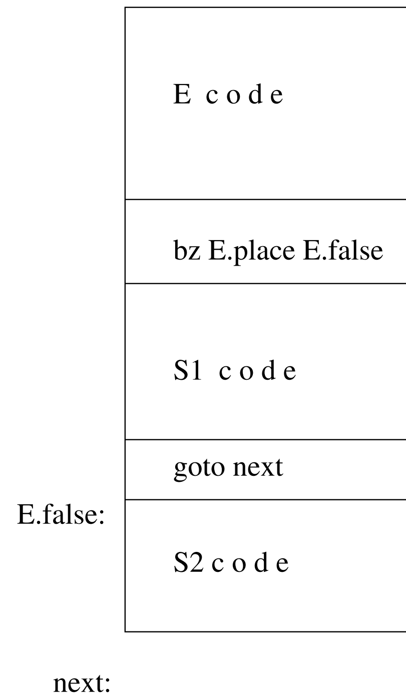
  - Explicit (syntactic) constructions for flow of control: if/while/for

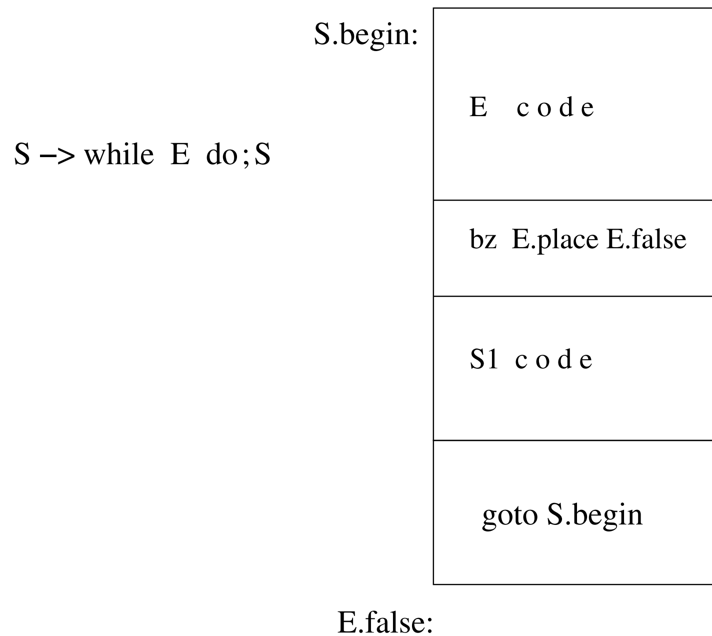
  - Implicit flow of control: function call (needs run-time organization)

S –> if E then S;



S –> if E then S
else S;





- The problem with these templates is that a forward label is generated *before* the next instruction which gets the label is generated. Since this rule cannot know the next instruction (if any), but needs the address of it, it can only specify the label.

eg.
```
if E1 then S1
 else S2;
if E2 then S3;
```

- Solutions:

  1. Build a syntax tree and make 2 passes over it. The second pass is top-down to fill in the labels. Uses inherited attributes.

  2. *Backpatching*: Generate branching instructions without labels. Then, when the statement with the label is generated, fill in the labels. This needs to introduce null productions just to mark points of label manipulation.

  3. If multiple labels per statement is allowed, we can generate labels for subsequent statements before they are generated. If the subsequent statement has its own label, there will be multiple labels.

A set of labels for the *same* stament forms an equivalence class, e.g.,

```
L1:L2:L3:  a:= b;
```

We can do a later pass over these equivalence classes and uniquely name them; this will give one label per statement. This is equivalent to backpatching in 2 passes.

Backpatching:

```
S -> if E then X S
     else S;

S -> while X E do X S;

X -> ε  {do something about list of labels}
```

2:

bz __1

goto --2

statement n−1

3:1:

statement n

Multiple labels:

```
S -> id := E ;      { S.code=E.code ||
                    2copy(id.place,E.place)}

S -> if E then S;   { t=newlabel();
                    S.code=E.code ||
                    bz E.place, t ||
                    S.code ||
                    label(t)}

S -> if E then S    { t1=newlabel();
  else S;           t2=newlabel();
                    S.code=E.code ||
                    bz E.place, t1 ||
                    S1.code ||
                    goto t2 ||
                    label(t1) ||
                    S2.code ||
                    label(t2)}
```

e.g.
```
a := E0;
if E1 then if E2 then S1 else S2;
if E3 then S3;
b:=E4;
```

E0 code

2copy(a,E0.place)

E1.code

bz E1.place,l2

E2 code

bz E2.place l1

S1 code

goto l2

l1:

S2 code

l2:l3:

E3 code

bz E3.place, l4

S3 code

l4: