A view of the object code Translation of (x+2)*(y-x)

The external representation (concrete syntax) is a three statement sequence below:

```
t1 := x.loc + #2
t2 := y.loc - x.loc
t3 := t1 * t2
```

The internal representation of a 3AC statement is simply **one** (possibly 32 bits) **word** with the following internal structure:

quadruple

opt	source1	source2	result

Hence, the following is the store map of the code

:=/+	x.loc	2	t1
:=/-	y.loc	x.loc	t2
:=/*	t1	t2	t3

What about the Translation of Commands: Assignment

How does it to do it ?

S |- el → (S,l)   S |- er → (S,v)

S |- el:=er → S[v/l]

(Semantics of assignment for expressions without)

|- el => ([|el|],ll) |- er => ([|er|],lr)

|- el :=er => [|el|]||[|er|]||ll:=lr

(Code Translation of assignments)

[11]C::= A
[12]C::= W
[13]A::= ide:=E {emit{ide.loc ":="E.loc}

In the command translation, we do not need any location in addition to the code that is anyway, generated by side effects

What about the Translation of Commands: Iterators

How does it to do it ?

(Semantics of while)

$$\frac{S \vdash e \rightarrow (S,true) \quad S \vdash c \rightarrow S' \quad S' \vdash while\ e\ do\ c \rightarrow S''}{S \vdash while\ e\ do\ c \rightarrow S''} \qquad \frac{S \vdash e \rightarrow (S,false)}{S \vdash while\ e\ do\ c \rightarrow S}$$

(Code Translation of while)

$$\frac{\vdash e \Rightarrow ([|e|],l) \quad S \vdash c \Rightarrow [|c|]}{\vdash while\ e\ do\ c \Rightarrow ([|e|]||\ldots||[|c|]||\ldots)}$$

How do code [|e|] and code [|c|] have to be combined together ?

What about the Translation of Combining Control Flows

[|E|]

if E.loc = #False goto nextC

[|C|]

[|Cs|]

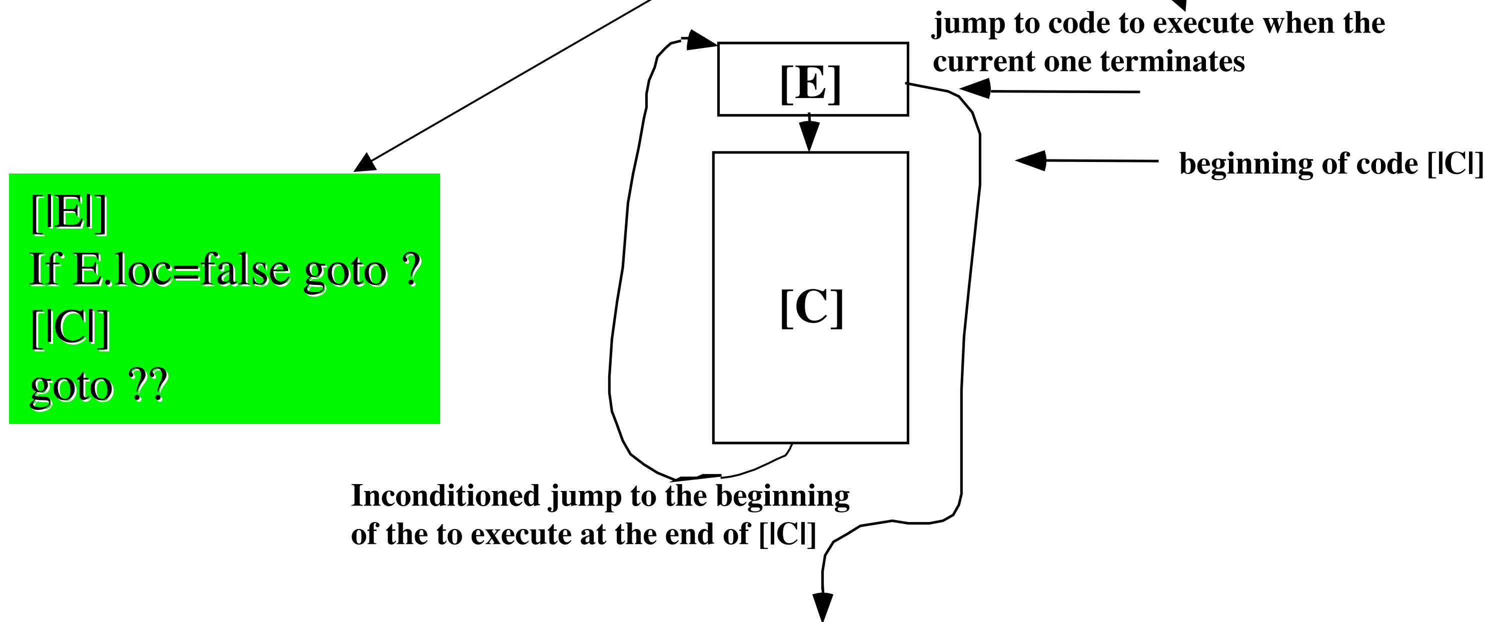
goto initC

[14]W::= while
E do {emit("if" E.loc "= #False goto" ?)}
C
Cs endw {emit("goto" ?)}

How do (instruction) locations nextC and initC have to be found and then, expressed in the translation actions of the grammar?

Commands that Change the Control Flow

[10]W::= while E do C endw



[10]W::= while E do {emit('if' E.loc '=' 'false' goto --)} C {emit(goto --)} endw

Using a counter *quad* in order to have a map of the instruction positions in the program

quad

- is a counter of the compiler
- contains the index of the next line of the emit-file to be allocated
- initially, is set to 0
- is incremented by 1, after each execution of emit

```
[10]W::= while {init:= quad} E do
               {emit('if' E.loc '=' 'false' goto --)} C
                                                       Cs
               {emit(goto init)} endw
```

Using a *Backpatching* in order to handle with jumps to unpredictable positions ahead

Backpatching and operation *BK*

- is used for references to the position R of each instruction I not yet generated
- It consists in:
  - **collecting** into a list L(R) all the positions of the instructions J that need such a position in the target operand (e.g.: goto R)
  - **letting** unspecifyied (i.e. marked "-") the value R in each J when J is right generated
  - **updating** each J in the list L(R) with the value found for R when the I eventually generated
  - Instruction update is expressed with action: **BK(L(R),R)**

```
[10]W::= while {init:= quad}
            E do {w.next:=mk-L(quad);
                    emit('if' E.loc '=' 'false' goto --)}
            C
            Cs {emit(goto init)}
           endw
```

What about the translation when C is, in turn, a command while

Where mk-L(a)=[a]; mk(a,L)=a::L; emptylist=[]; append=+

Translation Invariant of Command the attribute *.next*

- Each command is possibly, a source of control transfer
- Hence, each command has an attribute *.next*
- *C.next* = target-uncompleted instructions of C, that have to be completed with the address of code, to execute when C terminates

```
[10]W::= while {init:= quad}
         E do {w.next:= mk-L(quad);
               emit("if"E.loc "= #false goto --")}
         C {BK(C.next, quad)}
         Cs {BK(Cs.next, init); emit(goto init)}
        endw
```

Revisiting all the Phases of the One-Pass, Compositional, Code Translation

Use of side-effects: emit

```
[10]W::= while E do {emit('if' E.loc '=' 'false' goto --)} C {emit(goto --)} endw
```

Use of offset for code locations: quad

```
[10]W::= while {init:= quad} E do
                {emit('if' E.loc '=' 'false' goto --)} C
                {emit(goto init)} endw
```

Translation Invariant for Commands: .next

```
[10]W::= while {init:= quad} E do
                {w.next:=mk-L(quad);
                 emit('if' E.loc '=' 'false' goto --)} C
                {emit(goto init)} endw
```

Use of Backpatching: BK(…)

```
[10]W::= while {init:= quad} E do
                {w.next:= mk-L(quad);
                 emit('if' E.loc '=' 'false' goto --)} C {BK(C.next, init);
                 emit(goto init)} endw
```

What about the Translation of Command Sequencing

How does it to do it ?

$$\frac{S \vdash c \rightarrow S' \quad S' \vdash cs \rightarrow S''}{S \vdash c\ cs \rightarrow S''}$$

(Semantics of code sequencing)

$$\frac{}{S \vdash \lambda_c \rightarrow S}$$

[5] **Cs_1** ::= **; C** **{BK(C.next, quad)}**
Cs_2 **{Cs_1.next:= Cs_2.next}**
[6] **Cs**::= ε **{Cs.next:= []}**
…

Boolean Expression:

Short Circuit Translation for Control Transfer

Code Translation of a boolean expression B is, in short circuit, producing code that, when executed, trasfers the control totwo distincts code addresses:

- **code to execute if B evaluates to true**
- **code to execute if B evaluates to false**

Hence Short Circuit based Translation of espressions uses a different Translation Invariant:

- **attribute *.true*: target-uncomplete for true case transfer**
- **attribute .false: target-uncomplete for false case transfer**

source (x<y) or z

object

```
if x.loc<y.loc goto true-point
if z.loc goto true-point
goto false-point
```

Currently unknown: To be completed with the effective addresses

Short Circuit Translation for Control Transfer

Hence Short Circuit based Translation of espressions uses a different Translation Invariant:
- **attribute *.true*: target-uncomplete for true case transfer**
- **attribute *.false*: target-uncomplete for false case transfer**

Eb::= **Eb1 or M Eb2**

Eb.true::= app(**Eb1**.true, **Eb2**.true),
Eb.false::= **Eb2**.false,
BK(**Eb1**.false, **M**.quad)

Eb::= **Eb1 and M Eb2**

Eb.true::= **Eb2**.true,
Eb.false::= app(**Eb1**.false, **Eb2**.false),
BK(**Eb1**.true, **M**.quad)

Where mk-L(a)=[a]; mk(a,L)=a::L; emptylist=[]; append=+

Short Circuit Translation for Control Transfer / 2

Eb::= **true**

Eb.true::= MK(quad, emptylist),
Eb.false::= emptylist,
emit('goto' _)

Eb::= **false**

Eb.true::= emptylist,
Eb.false::= MK(quad, emptylist),
emit('goto' _)

Eb::= **ide**

Eb.true::= MK(quad, emptylist),
E.false::= MK(quad+1, emptylist),
emit('if' ide.loc '= #true goto' _),
emit('goto' _)

Where mk-L(a)=[a]; mk(a,L)=a::L; emptylist=[]