Goto Label and Switch

Implementing Simula Goto Statement
in Java using Exception handling and
ByteCode Engineering.

Simula Standard says:

A goto-statement interrupts the normal sequence of operations, by defining its successor explicitly by the value of a designational expression (i.e. a program point). Thus the next statement to be executed is the one at this program point.

The program point referenced by the designational expression must be visible at the goto-statement

Java does not support labels like Simula. The Java Virtual Machine (JVM), however, has labels.
A JVM-label is simply a relative byte-address within the byte-code of a method.

This implementation will generate Java-code which is prepared for Byte Code Engineering.
See chapter .....

Suppose a Simula program containing labels and goto like this:

```
Begin
  L:  ...
      goto L;
      ...
 End;
```

This will be coded as:

```
public final class adHoc00 extends BASICIO$ {
    public int prefixLevel() { return(0); }
    final LABQNT$ L=new LABQNT$(this,1); // Local Label #1=L
    public adHoc00(RTObject$ staticLink) {
        super(staticLink);
        BBLK();
        BPRG("adHoc00");
    } // End of Constructor

    // SimulaProgram Statements
    public RTObject$ STM() {
        adHoc00 THIS$=(adHoc00)CUR$;
        LOOP$:while(JTX$>=0) {
            try {
                JUMPTABLE$(JTX$); // For ByteCode Engineering
                // Statements ....
                LABEL$(1); // L
                // Statements ....
                throw(L); // GOTO EVALUATED LABEL
                // Statements ....
                break LOOP$;
            }
            catch(LABQNT$ q) {
                CUR$=THIS$;
                if(q.SL$!=CUR$) {
                    CUR$.STATE$=OperationalState.terminated;
                    throw(q); // Re-throw exception for non-local Label
                }
                JTX$=q.index; continue LOOP$; // GOTO Local L
            }
        }
        EBLK();
        return(null);
    }
}
```

Label Quantities.

At source line 3. the label 'L' is declared like this:

```
final LABQNT$ L=new LABQNT$(this,1); // Local Label #1=L
```

Where LABQNT$ is defined by:

```
public final class LABQNT$ extends RuntimeException
{ public RTObject$ SL$; // Static link, block in which the label is defined.
  public int index; // Ordinal number of Label within its Scope(staticLink).

  public LABQNT$(RTObject$ SL$,int index)
  { this.SL$=SL$; this.index=index; }
}
```

A goto-statement is simply coded as:

```
throw(L); // GOTO EVALUATED LABEL
```

And this exception is caught and tested (lines 23 - 38) throughout the operating chain. If the label does not belong to this block instance the exception is re-thrown.

In the event of no matching block instances the exception is caught by an UncaughtExceptionHandler like this:

```
public void uncaughtException(Thread thread, Throwable e) {
    if(e instanceof LABQNT$) {
        // POSSIBLE GOTO OUT OF COMPONENT
        RTObject$ DL=obj.DL$;
        if(DL!=null && DL!=CTX$) {
            DL.PENDING_EXCEPTION$=(RuntimeException)e;
            DL.resumeThread();
        } else {
            ERROR("Illegal GOTO "+e);
            ...
        } else ...
 }
```

Thus, when a QPS-component is left we raise the PENDING_EXCEPTION flag and resume next operating component. The resume-operations will re-trow the exception within its Thread.

For further details see the source code of RTObject.java

Byte Code Engineering.

The ObjectWeb ASM Java bytecode engineering library from the OW2 Consortium is used to modify the byte code to allow very local goto statements.

They say:

ASM is an all purpose Java bytecode manipulation and analysis framework. It can be used to modify existing classes or to dynamically generate classes, directly in binary form. ASM provides some common bytecode transformations and analysis algorithms from which custom complex transformations and code analysis tools can be built. ASM offers similar functionality as other Java bytecode frameworks, but is focused on performance. Because it was designed and implemented to be as small and as fast as possible, it is well suited for use in dynamic systems (but can of course be used in a static way too, e.g. in compilers).

More info at: https://asm.ow2.io and https://www.ow2.org/

To easily modify the code, the Simula Compiler generates certain method call in the .java file:

```
 - LABEL$(n); // Label #n
```

This method-call is used to signal the occurence of a Simula Label. The byte-code address is collected and some instruction are removed. The parameter 'n' is the label's ordinal number.

I.e. Try to locate the instruction sequence:

```
PREV-INSTRUCTION
ICONST n
INVOKESTATIC LABEL$
NEXT-INSTRUCTION
```

Pick up the number 'n', remember address and remove the two middle instruction.

```
 - JUMPTABLE$(JTX$); // For ByteCode Engineering
```

This method-call is a placeholder for where to put in a Jump-Table.

Try to locate the instruction sequence:

```
PREV-INSTRUCTION
GETFIELD JTX$
INVOKESTATIC JUMPTABLE$
NEXT-INSTRUCTION
```

And replace it by the instruction sequence:

```
PREV-INSTRUCTION
GETFIELD JTX$
TABLESWITCH ... uses the addresses collected for labels.
NEXT-INSTRUCTION
```

`For further details see the source code of ByteCodeEngineering.java`

Goto virtual label

```
BEGIN

   class A; virtual: label L;
   begin
      goto L;
   end;

   A class B;
   begin
      L: outtext("OK");
   end;

   ref(B) x;
   x:-new B;

END;
```

Men dette gikk ikke fordi statement-koden til klassen B ikke ligger på Java's dynamiske kjede når goto L blir utført.

Årsaken ligger i måten konkatenering av sub-klasser ble gjort:

Konkatenering av klasser ble gjort ved å bygge en datastruktur av statement-kodene noe slikt:

[statementene før inner] [inner] [statementene etter inner]

Hvor inner er kall på en tilsvarende struktur.

Ved en goto virtual label må jeg først følge inner-pekerne helt til bunns.
Der kaster jeg en passende exception og da har jeg fått den virtuelle labelen inn på Java's dynamiske kjede.

Siden jeg i det generelle tilfellet ikke kan vite om at label-expression evaluerer til en virtuell så må jeg gjøre dette for alle goto i klasser.

Jeg implementerte dette og koden ble større og større og mer og mer komplisert.

Til slutt valgte jeg en radikal løsning: Statement-koden til sub-klasser konkateneres eksplisitt, dvs. Koden til super-klassene kopieres inn i statement-koden.

Dermed var implementering av virtuelle labeler (og switcher) trivielt.

Kostnaden er kopiering av kode.

Jeg har sett litt på hvor mye dette utgjør:

- Alle system-klassene ble faktisk mindre.
  Spesiellt da jeg manuellt gjorde optimaliseringen som er beskrevet nedenfor.

- Sub-klasser av klassen DEMOS ble noe større, max 100 Java-linjer.
  Jeg har ikke eksakte mål men dette er helt ubetydelig på dagens maskiner.
  Det meste av denne koden var initiering av globale variable, og det kan jo gjøres annerledes.

En mulig optimalisering:

Vi kan samle all koden som kommer før første label eller **inner** i en egen metode 'init'
Og så legge ut et kall på 'init' i den dupliserte koden.

For DEMOS ville dette bety en reduksjon i duplisert kode på ca. 90 %

Men er det verdt det ?

Eksempel: En typisk sub-klasse av Process

```
Process class Person; begin
      while true do begin
         Hold(Normal(12, 4, U));
         fittingroom1.request;
         Hold(Normal(3, 1, U));
         fittingroom1.leave;
      end;
end;
```

```
public FittingRoom$Person STM$() {
    // Linkage: No code before inner
    // Link:    No code before inner
    detach();   // Process: Before inner
    // Person:  Code before inner
    while(true) {
         hold(normal(12,4,U));
         new request(fittingRoom1);
         hold(normal(3,1,U));
         new leave(fittingRoom1);
    }
    // Person:  No code after inner
    terminate(); // Process: After inner
    // Link:    No code after inner
    // Linkage: No code after inner
    EBLK();
    return(this);
}
```