Conventional scoping of registers — An experiment in ε_{χ}TEX

Gerd Neugebauer
In Lerchelsböhl 5
64521 Groß-Gerau (Germany)
gene (at) gerd-neugebauer dot de
www.gerd-neugebauer.de

Abstract

TEX provides groups as a means to restrict the visibility of registers. This construction is well known in the TEX world but does not coincide with groups as known from other programming languages. If we refrain from storing the register value in a global array we can come to the alternate solution of storing it in the control sequence used to access it. With this variant we can provide a means to define an arbitrary number of registers which follow the same scoping rules as variables in Pascal-like languages.

ε_{χ}TEX is a reimplementation of TEX in Java. It is developed with extensibility and configurability in mind. The idea of an alternative storage for registers can be implemented in ε_{χ}TEX as an extension. We show which steps are required for such an implementation. In this manner the extensibility of ε_{χ}TEX is demonstrated.

1 Registers and scoping

`plain.tex` provides macros to handle the allocation of registers. For this document we want to restrict our considerations to count registers. The other register types can be handled analogously. Here the macro `\newcount` can be used to allocate a new count register:

```
\newcount\abc
{\abc = 42
 \showthe\abc
}
```

In TEX any changes to registers are recorded. When the group closes, the old values are restored. Thus any changes to registers in a group are automatically local. This can be overwritten with the keyword `\global`.

Let us have a look at the same construction in another programming language. As an example we use Java. The same considerations hold for many languages of the Pascal family.

```
{ int abc = 42;
  System.out.println(abc);
}
```

The grouping reduces the scoping of the variable abc. It is defined within the group and not visible outside. If a variable with the same name is defined before the group then this variable is hidden by the new definition in the group.

The explicit declaration of the local variable in Java arranges things so that the new variable is activated and any previous declaration is hidden.

2 Storage in TEX

Coming back to TEX an alternative interpretation comes to mind. Whenever a register is modified in a group then an automatic declaration of a new variable is introduced and initialized.

One way to come closer to conventional programming languages with TEX would be to introduce typed variables following the conventional rules for scoping and initializing.

TEX stores the values of registers in TEX memory. This memory is written to the format file when a `\dump` is performed. Besides the register values, (macro) code is stored in TEX memory.

All we need is a primitive which behaves like a count register but stores the value somewhere else — making it accessible via the primitive only.

3 ε_{χ}TEX

The ε_{χ}TEX project (→ http://www.extex.org) has the aim to produce a reimplementation of TEX. The implementation language for this reimplementation is Java. The major design decisions put modularity and configurability into the forefront.

As one consequence ε_{χ}TEX is assembled out of many components. Those components provide defined interfaces. This makes it simple to write replacements for existing components and provide new

components to extend the system. This extensibility makes it easy to experiment to some extent with new ideas. In the following setions we will see one example of such an experiment.

$\varepsilon\chi$TEX is currently under development. Even though large pieces are in place, $\varepsilon\chi$TEX is not yet ready for production. Any help to get things finished is very welcome. If you are interested in participating in $\varepsilon\chi$TEX development, contact the developers on the developer list, which can be found via the $\varepsilon\chi$TEX web site.

4 Writing a new primitive for $\varepsilon\chi$TEX

According to our considerations we want to have a new primitive which behaves like a count register but stores the value within the code and not in the context. In addition we need a primitive `\integer` to dynamically create such integers. Then we can write the following TEX code:

```
{\integer \abc = 42
 \showthe\abc
}
```

First we start with implementing the code for the count-equivalent. This code needs to have several properties to behave like a count register:

- It needs to assign a new value when executed. This means that

  ```
  \abc=123
  ```

  works if `\abc` has the meaning of the new primitive.
- It needs to act as an assignment; this means that `\afterassignment` has to be taken into account. This mean its token is expanded after the assignment has taken place.
- It needs to be advanceable. This means that the following works:

  ```
  \advance\abc by 123
  ```

- It needs to be multiplyable. This means that the following works:

  ```
  \multiply\abc by 123
  ```

- It needs to be divideable. This means that the following works:

  ```
  \divide\abc by 123
  ```

- It needs to provide the count value upon request. This means that the following works:

  ```
  \count0=\abc
  ```

- It needs to provide value for primitives `\the` and `\showthe`. This means that the following works:

  ```
  \showthe\abc
  ```

- It needs to expand to the tokens making up its value.

5 Providing a definition

To start with we create a new class. This class lives in a package named `extex.tutorial`. In addition we use a bunch of imports from $\varepsilon\chi$TEX. Since the imports are usually filled in by the IDE, we omit them (like the comments which are assumed to be filled in by the reader).[1]

```
package extex.tutorial;

import org.extex.core.count.Count;
// a bunch more imports omitted
```

Next we declare the class. It is derived from an abstract base class which takes care of the assignment. Each of the properties we want to have is declared with the help of an interface. `Advanceable` describes that the primitive can be used after the primitive `\advance`, `Divideable` describes that the primitive can be used after the primitive `\divide` and so on. Each of these interfaces contains a single method which needs to be implemented.

```
public class IntPrimitive
        extends
            AbstractAssignment
        implements
            Advanceable,
            Divideable,
            Multiplyable,
            CountConvertible,
            Theable,
            ExpandableCode {
```

Since we want to store a count value with the code we first create a private field. The data type `Count` encapsulates a count value. It has the methods to access and manipulate it. In its core it contains a `long` value to store a number in.

```
private Count value = new Count(0);
```

But before we come to implement the interfaces we have to define a constructor. The constructor

[1] To be honest, the exact package structure of $\varepsilon\chi$TEX is subject to some changes until the final version 1.0 is released.

takes one argument — the name of the primitive — and passes it to the constructor of the super-class.

```
public IntPrimitive(String name) {
    super(name);
}
```

Now we can start with the first method `assign`. It takes four parameters with the following interfaces:

`Flags` contains the indicators for the prefix arguments like `\global`. The primitive can consume the flags and react differently upon their values. Since our primitive does not use prefixes this argument is simply ignored.

`Context` contains the equivalent to the TEX memory — anything contributing to the state of the interpreter is stored here. The `Context` is also stored in a format when `\dump` is invoked.

`TokenSource` provides access to the scanner and the parsing routines. It can be used to acquire further tokens or even higher order entities.

`Typesetter` contains the typesetter of the system. The typesetter produces nodes which might be stored in boxes and finally sent to the backend. The primitive can send characters or instructions to the typesetter or simply request some information from it.

We will see these parameters again with each of our methods.

```
public void assign(Flags prefix,
          Context context,
          TokenSource source,
          Typesetter typesetter)
        throws InterpreterException {

    source.getOptionalEquals(context);
    Count newValue = CountParser.parse(
        context, source, typesetter);
    value.set(newValue);
}
```

The implementation first consumes an optional equal sign and then parses a following count value. Finally we can set the internal count to this new value.

Assume that we have assigned the new primitive to the control sequence `\abc` — we will see the details later. Then we can do the following:

```
\abc = 1234
```

This simply assigns a new value to the variable. But we have also used the infrastructure of an assignment. Thus the tokens stored in the token register `\afterassignment` are inserted after the assignment:

```
\afterassignment=\x
\abc = 1234
\y
```

Right now we can assign a new value to the variable. Since we want to see what we have done, we implement the method `the` which converts the value back into tokens to be used by the primitives `\the` and `\showthe`.

```
public Tokens the(Context context,
        TokenSource source,
        Typesetter typesetter)
      throws InterpreterException,
            CatcodeException,
            ConfigurationException {

    return context.getTokenFactory().
        toTokens(value);
}
```

The main task of creating a list of tokens is provided by a token factory. This is an application of the factory pattern. The factory is attached to the context and can be retrieved from it.

Next we have to take care of `\advance`. In $\varepsilon\chi$TEX the implementation of `\advance` decouples the operation from the implementation of the primitive. Thus it is possible to add further primitives which can be used after `\advance`. This goal is reached with the help of the interface `Advanceable`. When the token has the meaning of code which implements this interface then the control is passed to the methods defined in the interface to carry out the operation. We use this feature to make our primitive applicable to `\advance`.

The method uses the parsing routines in $\varepsilon\chi$TEX to acquire the optional keyword `by` and the value for a count register. This value is added to the variable stored in this primitive.

```
public void advance(Flags prefix,
          Context context,
          TokenSource source,
          Typesetter typesetter)
        throws InterpreterException {

    source.getKeyword(context, "by");
    Count by = CountParser.parse(
          context,
          source,
          typesetter);
    value.add(by);
}
```

The same technique used for `\advance` is used for `\divide` as well. Thus we just have to implement the associated interface `Divideable` and provide the following method:

```
public void divide(Flags prefix,
       Context context,
       TokenSource source,
       Typesetter typesetter)
     throws InterpreterException {

   source.getKeyword(context, "by");
   Count by = CountParser.parse(
         context,
         source,
         typesetter);
   value.divide(by);
}
```

And once again the same trick for `\multiply`: We implement the interface `Multiplyable` and provide the following method:

```
public void multiply(Flags prefix,
       Context context,
       TokenSource source,
       Typesetter typesetter)
     throws InterpreterException {

   source.getKeyword(context, "by");
   Count by = CountParser.parse(
         context,
         source,
         typesetter);
   value.multiply(by);
}
```

Converting into a count value is expressed with the interface `Countconvertible` which has one method `convertCount`. This method delivers the count value as `long`. Since we have the variable in our private field we can just take the value from there.

```
public long convertCount(
       Context context,
       TokenSource source,
       Typesetter typesetter)
     throws InterpreterException {

   return value.getValue();
}
```

Finally we provide a means to use the primitive in an expandable context. When tokens are expanded — in contrast to executed — we simply push the tokens representing the value to the token source. Thus they are read and processed afterwards.

```
public void expand(Flags prefix,
       Context context,
       TokenSource source,
       Typesetter typesetter)
     throws InterpreterException {

   try {
      source.push(
         context.getTokenFactory().
            toTokens(value));
   } catch (CatcodeException e) {
      throw new InterpreterException(e)
   }
}
```

The method is slightly complicated by the handling of an exception which might come from the creation of the tokens. This exception is simply remapped and passed upwards.

This is all we need to do to implement the new primitive.

```
}
```

6 Putting things into place for testing

Now we are finished writing our new primitive as a Java class. But how can we make use of it? First of all we have to compile it with a Java compiler and put it into a jar — say, `abc.jar`. $\varepsilon\chi$TEX is installed in a directory. This installation directory contains a subdirectory named `lib`. All jars contained in this directory are automatically considered when classes are loaded. Thus we put `abc.jar` into this directory.

Next we make use of a quick extension mechanism to try out our fine new primitive. Later we will use the configuration mechanism of $\varepsilon\chi$TEX for this purpose. But now we simply use the dynamic extension mechanism which allows us to bind some Java code to a primitive. To do so we need to load the unit `jx`. Units in $\varepsilon\chi$TEX are collections of primitives. For instance there is a unit `tex` containing the TEX primitives.

One of the primitives contained in $\varepsilon\chi$TEX — i.e. in the unit `extex` — is the primitive `\ensureloaded`. It takes one argument in braces which is the name of a unit and loads this unit if has not yet been loaded into the interpreter.

This primitive is used now to load the unit `jx`:

```
\ensureloaded{jx}
```

After the unit `jx` has been loaded we can make use of the primitive `\javadef` provided by this unit. This primitive is similar to the primitive `\def`. It takes a control sequence and a list of tokens enclosed

in braces. The control sequence gets a new meaning. This meaning is determined by the Java class named in the tokens argument:

```
\javadef\abc{extex.tutorial.IntPrimitive}
```

Now we can use the primitive `\abc` as shown above. This is enough for testing. Nevertheless it is discouraged since it uses an implementation specific extension. The recommended way is to use the configuration facility described later.

7 Defining new variables

The definition of each new variable with `\javadef` is a little bit clumsy. Our original plan was to define any new variable with `\integer`. It takes a control sequence and the initial value. This can be accomplished with a small definition of the following kind:

```
\def\integer#1{%
  \javadef#1{extex.tutorial.IntPrimitive}%
  #1}
```

This approach works but has the disadvantage that the resulting macro does not interact properly with `\afterassignment`. The primitive `\javadef` is an assignment. Thus the afterassignment token would be inserted just after the definition but before the initial value has been read.

To overcome this problem and gain some more insight into the definition of primitives in εχTEX we implement this primitive in Java as well.

The class itself is started as shown before. Since the task is much simpler we do not need to declare a lot of implemented interfaces.

```
package extex.tutorial;

// a bunch of imports omitted

public class IntDef
       extends AbstractAssignment {
```

The constructor propagates the name to the super class — as before.

```
public IntDef(String name) {
   super(name);
}
```

Finally we have to implement the `assign` method. Here we can make use of the `TokenSource` to acquire a control sequence. Now we create a new instance of our class `IntPrimitive`. The argument is the name of the variable. This name is extracted from the control sequence token.

Now we can use the method `assign` of this new instance to assign the initial value. Finally we bind the new instance to the control sequence token. This binding makes use of an optional prefix argument `\global`. The prefix is read and cleared in one step. The clearing avoids an error message about unused prefix arguments.

The `\global` prefix allows us to define a global variable — even within a group. This extension was not on our initial agenda, but is easily implemented.

```
public void assign(Flags prefix,
         Context context,
         TokenSource source,
         Typesetter typesetter)
       throws InterpreterException {

   CodeToken cs =
       source.getControlSequence(
           context,
           typesetter);
   IntPrimitive code =
       new IntPrimitive(cs.toString());
   code.assign(Flags.NONE,
               context,
               source,
               typesetter);
   context.setCode(cs,
                   code,
                   prefix.clearGlobal());
}
```

Now we are finished and can use the primitive.

```
}
```

We have postponed the configuration of εχTEX until we have the primitive. Now we can fill this omission.

8 Configuring εχTEX

The encouraged way of extending εχTEX is by configuring a new unit. The configuration of a unit is an XML file following a particular schema. The outer tag is `unit`. It can have attributes. The mandatory attribute we are using is the attribute `name` which is used to specify the name.

As an inner tag we are using `primitives`. Inside this tag all additional primitives of this unit are listed with `define` specifications. The defines need attributes. The attribute `name` specifies the name of the control sequence to assign the definition to. The attribute `class` specifies the Java class. This class needs to implement the interface `Code`. This class is instantiated and bound to the control sequence.

The configuration file `tutorial.xml` has the following contents:

```
<unit name="tutorial">
  <primitives>
```

```
    <define name="integer"
            class="extex.tutorial.IntDef"/>
  </primitives>
</unit>
```

We have placed the compiled Java files in a jar. The configuration file `tutorial.xml` has to be placed in the same jar file. To be found, it has to be placed in a certain package. This is the package `config.unit`. Now we can load it like we have done with the unit `jx`:

```
\ensureloaded{tutorial}
```

9 Aliasing variables

With the variables introduced here we can use `\let` to create aliases for a variable. `\let` creates a new binding for a control sequence to the same code as an existing control sequence. With our implementation in mind it is immediately apparent that a modification of one variable at the same time also modifies all aliased variants. This is illustrated in the following example:

```
\integer\x=42
\let\y=\x
\x=123
\showthe\y
```

In this code `\x` and `\y` share the same content. After assigning 123 to `\x` this value also shows up when printing `\y`.

This trick can be used to access a variable which is hidden by a local variable. In this case you can make an alias before defining the local variable:

```
\integer\x=42
% . . .
{\let\y=\x
 \integer\x=123
 \showthe\y
}
```

10 Variables and name spaces

In [1], namespace support for ε-TeX was presented. Namespaces primarily act on primitives. This collides with the access to registers via one primitive — for instance `\count` for all count registers. The allocation macro `\newcount` from `plain` can be used to assign a control sequence to a certain count register. This control sequence is subject to the name space visibility. Nevertheless the control sequence can be bypassed.

With the variables introduced in this paper we can overcome this deficiency. The variables introduced interact in a natural way with the namespace concept of ε-TeX.

11 Conclusion

We have seen an alternate way of defining variables in ε-TeX. The scoping follows the rules of conventional programming languages. In contrast to registers, the number of variables is limited only by the memory available.

The implementation for ε-TeX has demonstrated the extensibility and configurability of the system. It has also shown that the proposed definition of variables leads to the desired results.

References

[1] Gerd Neugebauer. Namespaces for ε-TeX. In Volker RW Schaa, editor, *Proceedings EuroTEX 2005*, pages 67–70, 2005.