Data Structures in TEX

Marek Ryćko

EuroBachoTEX, May 2, 2007

data structures

TEX is a programming language

TEX is a programming language specialized in typesetting

algorithms + data structures
= programs

algorithms + data structures
= programs
Nicolaus Wirth

programming is expressing
our recipies
using data structures

no data structures
= no programs

almost no data structures
= almost no programs

generating energy by uniting

TEX + good programming tools

TEX + data structures

what are data structures?

abstract explanation

proper level of abstraction

proper path of abstraction

easier

easier to understand

easier to develop

easier to use

what is abstraction?

1 apple + 1 apple

$$1 \text{ apple} + 1 \text{ apple} = 2 \text{ apples}$$

$$1 \text{ table} + 1 \text{ table} = 2 \text{ tables}$$

too specific

too complicated

abstract from
the nature of the “thing”

$$1 + 1 = 2$$

another path of abstraction

abstract from
the number of the “things”

some apples + some apples
= some apples

true but not very useful

another level of abstraction

abstract from
the number of “things”
and the nature of “things”

$$\text{some things} + \text{some things} = \text{some things}$$

true but even less useful

so abstract explanation of...

what are data structures?

structures

to hold data

simple containers for data

and means to combine them

simple data containers

atoms

no internal structure

```
salt = "sugar"
```

```
\def\salt{sugar}
```

```
\newtoks\salt
\salt = {sugar}
```

```
\makeatom\salt{sugar}
```

```
\makeatom\water{air}
```

```
\makepair\sea\salt\water
```

\makenil\universe

abstract constructors

define a set of objects

`nil` is an object

for each data d
`atom`(d) is an object

for all objects *o1* and *o2*
`pair(`*o1*`, `*o2*`)` is an object

abstract destructors

`isnil(`*object*`)`

`isatom(`*object*`)`

`ispair(`*object*`)`

`left(`*object*`)`

`right(`*object*`)`

`value(`*object*`)`

clean approach:

separate specification
from implementation

access to data
only through the operations

the user is to think
in terms of the specification

the implementation
may change

next level

lists of atoms

abstract definition

abstract constructors

define a set of atoms

for each data d
`atom`(d) is an atomic object

define a set of lists of atoms

`empty` is a list

for each atom a
and each list l
`append`(a, l) is a list

abstract deconstructors

`isempty(`*list*`)`

`head(`*list*`)`

`tail(`*list*`)`

the user is to think
in terms of the specification

the specification

specify
the list operations

specify
the list operations
in terms of
the pair operations

empty is nil

append is pair

isempty is isnil

head is left

tail is right

an implementation

one of possible implementations

implement the list operations

implement
the list operations
in terms of
the pair operations

empty is nil

append is pair

isempty is isnil

head is left

tail is right

the implementor has a freedom

use a “pair” module

or

not to use a “pair” module

furhter specifications

furhter specifications and implementations

similarly simple

lists of arbitrary objects inluding pair constructs and lists

one-liners

!!!

rectangular arrays of objects
are just lists of lists

current situation

if we had an implementation of pair objects in TEX

then we would have
an implementation
of *all* necessary data structures

back from abstraction

TEX implementation of the pair operations

how to store data inside TEX?

how to store *nil*, atoms and paris?

easy to implement operations

constructors

destructors

time for design decisions

values of atoms = token sequences

objects are stored in macros

objects are stored
with universal operations

objects are alive

nil

nil is stored in a macro

value *nil* stored in a variable `\N`

is equivalent to the result of:

```
\def\N{\nil}
```

operation to assign the value *nil*
to an arbitrary variable:

```
\def\makenil#1{\def#1{\nil}}
```

the operation `\makenil` applied to a variable `\N`:

```
\makenil\N
```

the result is equivalent to:

```
\def\N{\nil}
```

at the moment of defining `\N`
the control sequence `\nil`
is not defined

the meaning of `\nil`
will be defined before using `\N`

atoms

atom containing value `abc`
stored in a variable `\A`

is equivalent to the result of:

```
\def\A{\atom{abc}}
```

operation to assign
the atomic value `abc`
to an arbitrary variable:

```
\def\makeatom
  #1% macro name
  #2% atom value
  {\def#1{\atom{#2}}}
```

the operation `\makeatom`
applied to a variable `\A`
and intended value `abc`
of the atom:

```
\makeatom\A{abc}
```

the result is equivalent to:

```
\def\A{\atom{abc}}
```

at the moment of defining `\A`
the control sequence `\atom`
is not defined

pairs

pair containing values
of two objects
(*nil-s*, atoms and/or pairs)
stored in a variable `\P`

is equivalent to the result of:

```
\def\P{\pair{...}{...}}
```

operation to assign
the pair value
to an arbitrary variable:

```
\def\makepair
  #1% macro name
  #2% left object
  #3% right object
  {\def#1{\pair{#2}{#3}}}
```

after defining variables `\A` and `\N`:

```
\makeatom\A{abc}
\makenil\N
```

we may define variable `\P`:

```
\makepair\P\A\N
```

the result is equivalent to:

```
\def\P
  {\pair{\atom{abc}}{\nil}}
```

```
\def\P
  {\pair{\atom{abc}}{\nil}}
```

```
\def\P
  {\pair{\atom{abc}}{\nil}}
```

constructors are already defined

and so is
the internal representation

time to define destructors

the general idea

1. define the meaning of `\nil`, `\atom` and `\pair`

2. execute the object

example

define `\left` as:
“get the left element
of a pair object”

`\left\L\P` means:
"define macro `\L`
as the left element
of pair `\P`"

if we defined
\A, \N and \P:

```
\makeatom\A{abc}
\makenil\N
\makepair\P\A\N
```

the resulting value of \P
is the same
as if it were defined as:

```
\def\P
  {\pair{\atom{abc}}{\nil}}
```

\left\L\P
is expected to assign \atom{abc}
to \L

implementation of `\left`:

1. define macro `\pair`

2. execute decomposed object

one-liner,
more readable in seven lines

```
\def\left#1#2{\def\pair##1##2{\def#1{##1}}#2}
```

```
\def\left#1#2{\def\pair
##1##2{\def#1{##1}}#2}
```

```
\def\left
  #1% result macro name
  #2% decomposed object
  {\def\pair
    ##1##2%
    {\def#1{##1}}%
  #2%
  }
```

```
\def\right
  #1% result macro name
  #2% decomposed object
  {\def\pair
    ##1##2%
    {\def#1{##2}}%
  #2%
  }
```

features

clean design

easy to understand

separation of specification and implementation

TEX dialect independent

TEX dialect independent
(uses only very low level
TEX operations)

efficient implementation using TEX's capabilities

very, very simple

it has always been there!