Lambda calculus in TEX

Victor Eijkhout

August 2004

1 Logic with TEX

1.1 Truth values, operators

We start by defining a couple of simple tools.

```
\def\Ignore#1{}
\def\Identity#1{#1}
\def\First#1#2{#1}
\def\Second#1#2{#2}
```

For example:

Taking first argument:

```
input : \First {first}{second}
output : first
```

Taking second argument:

```
input : \Second {first}{second}
output : second
```

We define truth values:

```
\let\True=\First
\let\False=\Second
```

and logical operators:

```
\def\And#1#2{#1{#2}\False}
\def\Or#1#2{#1\True{#2}}
\def\Twiddle#1#2#3{#1{#3}{#2}}
\let\Not=\Twiddle
```

Explanation: `And` x y is y if x is true, false is x is false. Since True and False are defined as taking the first and second component, that gives the definition of `And` as above. Likewise `Or`.

To test logical expressions, we attach `TF` to them before evaluting; that was `\True TF` will print `T`, and `\False TF` will print `F`.

Let us test the truth values and operators:

True takes first of TF:
```
input : \True
output : T
```
False takes second of TF:
```
input : \False
output : F
```
Not true is false:
```
input : \Not \True
output : F
```
And truth table TrueTrue:
```
input : \And \True \True
output : T
```
And truth table TrueFalse:
```
input : \And \True \False
output : F
```
And truth table FalseTrue:
```
input : \And \False \True
output : F
```
And truth table FalseFalse:
```
input : \And \False \False
output : F
```
Or truth table TrueTrue:
```
input : \Or \True \True
output : T
```
Or truth table TrueFalse:
```
input : \Or \True \False
output : T
```
Or truth table FalseTrue:
```
input : \Or \False \True
output : T
```
Or truth table FalseFalse:
```
input : \Or \False \False
output : F
```

1.2 Conditionals

Some more setup. We introduce conditionals

```
\def\gobblefalse\else\gobbletrue\fi#1#2{\fi#1}
\def\gobbletrue\fi#1#2{\fi#2}
\def\TeXIf#1#2{#1#2 \gobblefalse\else\gobbletrue\fi}
\def\IfIsPositive{\TeXIf{\ifnum0<}}
```

with the syntax

```
\TeXIf <test> <arg>
```

We test this:

Numerical test:
```
input : \IfIsPositive {3}
output : T
```
Numerical test:
```
input : \IfIsPositive {-2}
output : F
```

1.3 Lists

A list is defined as a construct with a head, which is an element, and a tail, which is another list. We will denote the empty list by `Nil`.

```
\let\Nil=\False
```

We implement a list as an operator with two arguments:

- If the list is not empty, the first argument is applied to the head, and the tail is evaluated;

- If the list is empty, the second argument is evaluated.

In other words

$$L\,a_1\,a_2 = \begin{cases} a_2 & \text{if } L = () \\ a_1(x)\,Y & \text{if } L = (x, Y) \end{cases}$$

In the explanation so far, we only know the empty list `Nil`. Other lists are formed by taking an element as head, and another list as tail. This operator is called `Cons`, and its result is a list. Since a list is a two argument operator, we have to make `Cons` itself a four argument operator:

```
% \Cons <head> <tail> <arg1> <arg2>
\def\Cons#1#2#3#4{#3{#1}{#2}}
```

Since `Cons#1#2` is a list, applied to `#3#4` it should expand to the second clause of the list definition, meaning it applies the first argument (`#3`) to the head (`#1`), and evaluates the tail (`#2`).

The following definitions are typical for list operations: since a list is an operator, applying an operation to a list means applying the list to some other objects.

```
\def\Error{{ERROR}}
\def\Head#1{#1\First\Error}
\def\Tail#1{#1\Second\Error}
```

Let us take some heads and tails of lists. As a convenient shorthand, a singleton is a list with an empty tail:

```
\def\Singleton#1{\Cons{#1}\Nil}
```

Head of a singleton:

```
input : \Head {\Singleton \True }
output : T
```

Head of a tail of a 2-elt list:

```
input : \Head {\Tail {\Cons \True {\Singleton \False
  }}}
output : F
```

We can also do arithmetic tests on list elements:

Test list content:

```
input : \IfIsPositive {\Head {\Singleton {3}}}
output : T
```

Test list content:

```
input : \IfIsPositive {\Head {\Tail {\Cons
  {3}{\Singleton {-4}}}}}
output : F
```

Exercise 1.

Write a function `\IsNil` and test with

```
\test{Detect NIL}{\IsNil\Nil}
\test{Detect non-NIL}{\IsNil{\Singleton\Nil}}
```

1.3.1 A list visualization tool

If we are going to be working with lists, it will be a good idea to have a way to visualize them. The following macros print a '1' for each list element.

```
\def\Transcribe#1{#1\TranscribeHT\gobbletwo}
\def\TranscribeHT#1#2{1\Transcribe{#2}}
```

1.3.2 List operations

Here are some functions for manipulating lists. We want a mechanism that takes a function f, an initial argument e, and a list X, so that

$$\texttt{Apply}\,f\,e\,X \Rightarrow f\,x_1\,(f\,x_2\,(\ldots(f\,x_n\,e)\ldots))$$

```
% #1=function #2=initial arg #3=list
\def\ListApply#1#2#3{#3{\ListApplyp{#1}{#2}}{#2}}
\def\ListApplyp#1#2#3#4{#1{#3}{\ListApply{#1}{#2}{#4}}}
```

This can for instance be used to append two lists:

```
\def\Cat#1#2{\ListApply\Cons{#2}{#1}}
```

For example:

Cat two lists:

```
input : \Transcribe {\Cat {\Singleton \Nil }{\Cons \Nil
  {\Singleton \Nil }}}
output : 111
```

From now on the `\Transcribe` macro will be implicitly assumed; it is no longer displayed in the examples.

1.4 Numbers

We can define integers in terms of lists: zero is the empty list, and to add one to a number is to `Cons` it with an empty list as head element. In other words,

$$n + 1 \equiv (0, n).$$

This defines the 'successor' function on the integers.

```
\let\Zero\Nil
\def\AddOne#1{\Cons\Nil{#1}}
```

Examples:

Transcribe zero:

```
input : \Zero
output :
```

Transcribe one:

```
input : \AddOne \Zero
output : 1
```

Transcribe three:

```
input : \AddOne {\AddOne {\AddOne \Zero }}
output : 111
```

Writing this many `\AddOne`s get tiring after a while, so here is a useful macro:

```
\newtoks\dtoks\newcount\nn
\def\ndef#1#2{\nn=#2 \dtoks={\Zero}\nndef#1}
\def\nndef#1{
  \ifnum\nn=0 \edef\tmp{\def\noexpand#1{\the\dtoks}}\tmp
  \else \edef\tmp{\dtoks={\noexpand\AddOne{\the\dtoks}}}\tmp
        \advance\nn by -1 \nndef#1
  \fi}
```

which allows us to write

```
\ndef\One1 \ndef\Two2 \ndef\Three3 \ndef\Four4 \ndef\Five5
\ndef\Seven7\ndef\Six6
```

et cetera.

It is somewhat surprising that, even though the only thing we can do is compose lists, the predecessor function is just as computable as the successor:

```
\def\SubOne#1{#1\Second\Error}
```

Predecessor of two:

```
input : \SubOne {\AddOne {\AddOne \Zero }}
output : 1
```

(If we had used `\Ignore` instead of `\Second` a subtle TEXnicality would come into play: the list tail would be inserted as `{#2}`, rather than `#2`, and you would see an `Unexpected }` error message.)

Some simple arithmetic: we test if a number is odd or even.

```
\def\IsEven#1{#1\IsOddp\True}
\def\IsOddp#1#2{\IsOdd{#2}}
\def\IsOdd#1{#1\IsEvenp\False}
\def\IsEvenp#1#2{\IsEven{#2}}
```

Zero even?:

```
input : \IsEven \Zero
output : T
```

Zero odd?:

```
input : \IsOdd \Zero
output : F
```

Test even:

```
input : \IsEven {\AddOne
  {\AddOne {\AddOne \Zero }}}
output : F
```

Test odd:

```
input : \IsOdd {\AddOne
  {\AddOne {\AddOne \Zero }}}
output : T
```

Test even:

```
input : \IsEven {\AddOne
  {\AddOne {\AddOne {\AddOne
  {\Zero }}}}}
output : T
```

Test odd:

```
input : \IsOdd {\AddOne
  {\AddOne {\AddOne {\AddOne
  {\Zero }}}}}
output : F
```

Exercise 2. Write a test `\IsOne` that tests if a number is one.

Zero:

```
input : \IsOne \Zero
output : F
```

One:

```
input : \IsOne \One
output : T
```

Two:

```
input : \IsOne \Two
output : F
```

1.4.1 Arithmetic: add, multiply

Above, we introduced list concatenation with `\Cat`. This is enough to do addition. To save typing we will make macros `\Three` and such that stand for the usual string of `\AddOne` compositions:

```
\let\Add=\Cat
```

Adding numbers:

```
input : \Add {\Three }{\Five }
output : 11111111
```

Instead of adding two numbers we can add a whole bunch

```
\def\AddTogether{\ListApply\Add\Zero}
```

For example:

Adding a list of numbers:

```
input : \AddTogether {\Cons \Two {\Singleton \Three }}
output : 11111
```

Adding a list of numbers:

```
input : \AddTogether {\Cons \Two {\Cons \Three
  {\Singleton \Three }}}
output : 11111111
```

This is one way to do multiplication: to evaluate 3×5 we make a list of 3 copies of the number 5.

```
\def\Copies#1#2{#1{\ConsCopy{#2}}\Nil}
\def\ConsCopy#1#2#3{\Cons{#1}{\Copies{#3}{#1}}}
\def\Mult#1#2{\AddTogether{\Copies{#1}{#2}}}
```

Explanation:

- If `#1` of `\Copies` is empty, then `Nil`.
- Else, `\ConsCopy` of `#2` and the head and tail of `#1`.
- The tail is one less than the original number, so `\ConsCopy` makes that many copies, and conses the list to it.

For example:

Multiplication:

```
input : \Mult {\Three }{\Five }
output : 111111111111111
```

However, it is more elegant to define multiplication recursively.

```
\def\MultiplyBy#1#2{%
  \IsOne{#1}{#2}{\Add{#2}{\MultiplyBy{\SubOne{#1}}{#2}}}}
```

Multiply by one:

```
input : \MultiplyBy \One \Five
output : 11111
```

Multiply bigger:

```
input : \MultiplyBy \Three \Five
output : 111111111111111
```

1.4.2 More arithmetic: subtract, divide

The recursive definition of subtraction is

$$m - n = \begin{cases} m & \text{if } n = 0 \\ (m-1) - (n-1) & \text{otherwise} \end{cases}$$

Exercise 3. Implement a function `\Sub` that can subtract two numbers. Example:

Subtraction:

```
input : \Sub \Three \Five
output : 11
```

1.4.3 Continuing the recursion

The same mechanism we used for defining multiplication from addition can be used to define taking powers:

```
\def\ToThePower#1#2{%
  \IsOne{#1}{#2}{%
    \MultiplyBy{#2}{\ToThePower{\SubOne{#1}}{#2}}}}
```

Power taking:

```
input : \ToThePower {\Two }{\Three }
output : 111111111
```

1.4.4 Testing

Some arithmetic tests. Greater than: if

$$X = (x, X'), \quad Y = (y, Y')$$

then $Y > X$ is false if $Y \equiv 0$:

```
\def\GreaterThan#1#2{#2{\GreaterEqualp{#1}}\False}
```

Otherwise, compare X with $Y' = Y - 1$: $Y > X \Leftrightarrow Y' \geq X$; this is true if $X \equiv 0$:

```
\def\GreaterEqualp#1#2#3{\GreaterEqual{#1}{#3}}
\def\GreaterEqual#1#2{#1{\LessThanp{#2}}\True}
```

Otherwise, compare $X' = X - 1$ with $Y' = Y - 1$:

```
\def\LessThanp#1#2#3{\GreaterThan{#3}{#1}}
```

Greater (true result):
```
input : \GreaterThan \Two
  \Five
output : T
```
Greater (false result):
```
input : \GreaterThan \Three
  \Two
output : F
```
Greater (equal case):
```
input : \GreaterThan \Two
  \Two
output : F
```
Greater than zero:
```
input : \GreaterThan \Two
  \Zero
output : F
```
Greater than zero:
```
input : \GreaterThan \Zero
  \Two
output : T
```

Instead of just printing 'true' or 'false', we can use the test to select a number or action:

Use true result:
```
input : \GreaterThan \Two \Five \Three \One
output : 111
```
Use false result:
```
input : \GreaterThan \Three \Two \Three \One
output : 1
```

Let's check if the predicate can be used with arithmetic.

$3 < (5-1)$:
```
input : \GreaterThan \Three {\Sub \One \Five }
output : T
```
$3 < (5-4)$:
```
input : \GreaterThan \Three {\Sub \Four \Five }
output : F
```

Equality:
```
\def\Equal#1#2{#2{\Equalp{#1}}{\IsZero{#1}}}
\def\Equalp#1#2#3{#1{\Equalx{#3}}{\IsOne{#2}}}
\def\Equalx#1#2#3{\Equal{#1}{#3}}
```

Equality, true:
```
input : \Equal \Five \Five
output : T
```
Equality, true:
```
input : \Equal \Four \Four
output : T
```
Equality, false:
```
input : \Equal \Five \Four
output : F
```
Equality, false:
```
input : \Equal \Four \Five
output : F
```
$(1+3) \equiv 5$: false:
```
input : \Equal {\Add \One
  \Three }\Five
output : F
```
$(2+3) \equiv (7-2)$: true:
```
input : \Equal {\Add \Two
  \Three }{\Sub \Two \Seven }
output : T
```

Fun application:

```
\def\Mod#1#2{%
  \Equal{#1}{#2}\Zero
    {\GreaterThan{#1}{#2}%
       {\Mod{#1}{\Sub{#1}{#2}}}%
       {#2}%
    }}
```

$\mathrm{Mod}(27, 4) = 3$:

```
input : \Mod \Four \TwentySeven
output : 111
```

$\mathrm{Mod}(6, 3) = 0$:

```
input : \Mod \Three \Six
output :
```

With the modulo operation we can compute greatest common divisors:

```
\def\GCD#1#2{%
   \Equal{#1}{#2}%
      {#1}%
      {\GreaterThan{#1}{#2}%             % #2>#1
         {\IsOne{#1}\One
            {\GCD{\Sub{#1}{#2}}{#1}}}%   % then take GCD(#2-#1,#1)
         {\IsOne{#2}\One
            {\GCD{\Sub{#2}{#1}}{#2}}}}}  % else GCD(#1-#2,#2)
```

GCD(27,4)=1:

```
input : \GCD \TwentySeven \Four
output : 1
```

GCD(27,3)=3:

```
input : \GCD \TwentySeven \Three
output : 111
```

and we can search for multiples:

```
\def\DividesBy#1#2{\IsZero{\Mod{#1}{#2}}}
\def\NotDividesBy#1#2{\GreaterThan\Zero{\Mod{#1}{#2}}}
\def\FirstDividesByStarting#1#2{%
  \DividesBy{#1}{#2}{#2}{\FirstDividesByFrom{#1}{#2}}}
\def\FirstDividesByFrom#1#2{\FirstDividesByStarting{#1}{\AddOne{#2}}}
```

$5|25$:

```
input : \DividesBy \Five \TwentyFive
output : T
```

$5 \not| 27$:

```
input : \DividesBy \Five \TwentySeven
output : F
```

$5 \not| 27$:

```
input : \NotDividesBy \Five \TwentySeven
output : T
```

$10 = \min\{i : i \geq 7 \wedge 5|i\}$:

```
input : \FirstDividesByFrom \Five \Seven
output : 1111111111
```

1.5 Infinite lists

So far, we have dealt with lists that are finite, built up from an empty list. However, we can use infinite lists too.

```
\def\Stream#1{\Cons{#1}{\Stream{#1}}}
```

Infinite objects:

```
input : \Head {\Tail {\Stream 3}}
output : 3
```

Infinite objects:

```
input : \Head {\Tail {\Tail {\Tail {\Tail {\Tail
  {\Stream 3}}}}}}
output : 3
```

Even though the list is infinite, we can easily handle it in finite time, because it is never constructed further than we ask for it. This is called 'lazy evaluation'.

We can get more interesting infinite lists by applying successive powers of an operator to the list elements. Here is the definition of the integers by applying the `AddOne` operator a number of times to zero:

```
% \StreamOp <operator> <initial value>
\def\StreamOp#1#2{\Cons{#2}{\StreamOp{#1}{#1{#2}}}}
\def\Integers{\StreamOp\AddOne\Zero}
\def\PositiveIntegers{\Tail\Integers}
\def\TwoOrMore{\Tail\PositiveIntegers}
```

Again, the `Integers` object is only formed as far as we need it:

Integers:

```
input : \Head {\Tail {\Integers }}
output : 1
```

Integers:

```
input : \Head {\Tail {\Tail {\Tail {\Tail {\Tail
  {\Integers }}}}}}
output : 11111
```

Let us see if we can do interesting things with lists. We want to make a list out of everything that satisfies some condition.

```
\def\ConsIf#1#2#3{#1{#2}{\Cons{#2}{#3}}{#3}}
\def\Doubles{\ListApply{\ConsIf{\DividesBy\Two}}\Nil\PositiveIntegers}
\def\AllSatisfy#1{\ListApply{\ConsIf{#1}}\Nil\PositiveIntegers}
\def\FirstSatisfy#1{\Head{\AllSatisfy{#1}}}
```

third multiple of two:

```
input : \Head {\Tail {\Tail \Doubles }}
output : 111111
```

old enough to drink:

```
input : \FirstSatisfy {\GreaterThan \TwentyOne }
output : 1111111111111111111111
```

We add the list in which we test as a parameter:

```
\def\AllSatisfyIn#1#2{\ListApply{\ConsIf{#1}}\Nil{#2}}
\def\FirstSatisfyIn#1#2{\Head{\AllSatisfyIn{#1}{#2}}}
```

:

```
input : \FirstSatisfyIn {\NotDividesBy {\FirstSatisfyIn
  {\NotDividesBy \Two }\TwoOrMore }} {\AllSatisfyIn
  {\NotDividesBy \Two }\TwoOrMore }
output : 11111
```

And now we can repeat this:

```
\def\FilteredList#1{\AllSatisfyIn{\NotDividesBy{\Head{#1}}}{\Tail{#1}}}
\def\NthPrime#1{\Head{\PrimesFromNth{#1}}}
\def\PrimesFromNth#1{\IsOne{#1}\TwoOrMore
  {\FilteredList{\PrimesFromNth{\SubOne{#1}}}}}
```

Third prime; spelled out:

```
input : \Head {\FilteredList {\FilteredList \TwoOrMore
  }}
output : 11111
```

Fifth prime:

```
input : \NthPrime \Four
output : 1111111
```

However, this code is horrendously inefficient. To get the 7th prime you can go make a cup of coffee, one or two more and you can go pick the beans yourself.

```
%\def\FilteredList#1{\AllSatisfyIn{\NotDividesBy{\Head{#1}}}{\Tail{#1}}}
\def\xFilteredList#1#2{\AllSatisfyIn{\NotDividesBy{#1}}{#2}}
\def\FilteredList#1{\xFilteredList{\Head{#1}}{\Tail{#1}}}
```

Fifth prime:

```
input : \NthPrime \Five
output : 11111111111
```

Contents