Eidgenössische Technische Hochschule Zürich
Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Federal Institute of Technology at Zurich

Michael Grossniklaus

Informatik 3

Winter Term 2001/2002

Week 2

Exercise 2.1:

1. To reduce $((\lambda z.(za))(\lambda x.x))$ we use the leftmost-outermost method presented in the lecture. In this case, this means to substitute z before x.

$$\begin{array}{lll l} & ((\lambda z.(za))(\lambda x.x)) & [(\lambda x.x)/z] & (\beta\text{-conversion}) \\ = & (\lambda x.x)a & [a/x] & (\beta\text{-conversion}) \\ = & a & & \end{array}$$

2. Again we will use the leftmost-outermost method. Doing so we will begin by substituting $(\lambda z.x)$ for y. Here we have to be careful because this substitution would introduce a new bound variable in the term $(\lambda x.yx)$ which is not allowed. Before perfoming the operation we will have to rename the variables of the above term to $(\lambda u.yu)$

$$\begin{array}{llll} & (\lambda y.(\lambda u.[u/x]yx)y)(\lambda z.x) & & (\alpha\text{-conversion}) \\ = & (\lambda y.(\lambda u.yu)y)(\lambda z.x) & [(\lambda z.x)/y] & (\beta\text{-conversion}) \\ = & (\lambda u.(\lambda z.x)u)(\lambda z.x) & [(\lambda z.x)/u] & (\beta\text{-conversion}) \\ = & (\lambda z.x)(\lambda z.x) & [(\lambda z.x)/z] & (\beta\text{-conversion}) \\ = & (\lambda z.x) & & \\ = & x & & \end{array}$$

3. The following term can be simplified in various ways. It is even possible to do some steps in parallel. Nevertheless we will use this example to show once more the use of the leftmost-outermost method, which will always yield a non-reducible expression if one can be found.

$$\begin{array}{llll} & (\lambda y.yy)((\lambda x.x)(\lambda z.z)) & [((\lambda x.x)(\lambda z.z))/y] & (\beta\text{-conversion}) \\ = & ((\lambda x.x)(\lambda z.z))((\lambda x.x)(\lambda z.z)) & [(\lambda z.z)/x] & (\beta\text{-conversion}) \\ = & (\lambda z.z)((\lambda x.x)(\lambda z.z)) & [((\lambda x.x)(\lambda z.z))/z] & (\beta\text{-conversion}) \\ = & ((\lambda x.x)(\lambda z.z)) & [(\lambda z.z)/x] & (\beta\text{-conversion}) \\ = & (\lambda z.z) & & \end{array}$$

Exercise 2.2:

The first two exercises found in the script do not involve any SML programming. Here are possible answers to the questions about the logical functions `andalso` and `orelse` and the operator precedence in SML.

Logical Functions SML uses **short-circuit evaluation** to determine the value of these logical functions. The short-circuit method only evaluates terms contained in a logical expression as long as the result of the entire formula is unknown. If the first term E_1 in an expression E_1 `orelse` E_2 is $true$, E_2 will **not** evaluated. Vice versa, if the first term E_1 in an expression E_1 `andalso` E_2 is $false$, E_2 will **not** be evaluated. To indicate this behaviour the functions have been named accordingly.

Operator Precedence in SML The operator precedence is defined as follows. The top operator has highest precedence, the bottom operators have lowest.

```
~
mod, div
*, /
+, -
```

Exercise 2.3:

1. The function `trunc` converts a real to an integer through truncation, i.e. taking only the integer part of the real. The difference to the system-defined function `floor` is, as we can see, the treatment of negative numbers. $floor(-1.5)$ returns -2, whereas $trunc(-1.5)$ yields -1.

   ```
   fun trunc x = if x < 0.0 then ~1 * floor(~x)
                            else floor(x);

   val trunc = fn : real -> int
   ```

2. The function `sign` computes the sign of an integer returning strings `"+"`, `"0"` and ”-” to indicate whether am integer is positive, zero or negative, respectively.

   ```
   fun sign 0 = "0"
     | sign x = if x < 0 then "-" else "+";

   val sign = fn : int -> string
   ```

3. The function `power2` returns `true` if a positive integer is a power of 2 and `false` otherwise.

   ```
   fun power2 0 = false
     | power2 1 = true
     | power2 n = (n mod 2 = 0) andalso power2(n div 2);

   val power2 = fn : int -> bool
   ```

Exercise 2.4:

1. The function `third` returns the third element of a list of integers. If there are less than three alements in the list, then it returns -1.

   ```
   fun third nil = ~1
     | third (st::nil) = ~1
     | third (st::nd::nil) = ~1
     | third (st::nd::rd::tl) = rd;

   val third = fn : int list -> int
   ```

2. The function append takes two lists and forms a new list by appending the second list to the first.

   ```
   fun append left nil = left
     | append nil right = right
     | append (hd::tl) right = hd::append tl right;

   val append = fn : 'a list -> 'a list -> 'a list
   ```

   Note: In SML there is a built-in operator `@` that concatenates two lists. Instead of `append(left, right)` it is possible to use `left @ right`.

3. The function tt reverse takes a list and returns a list with the same elements, but with the order of the elements reversed.

   ```
   fun reverse nil = nil
     | reverse (hd::tl) = append (reverse tl) [hd];

   val reverse = fn : 'a list -> 'a list
   ```

Exercise 2.5:
The following function will, when given two dates – a person's birthdate and today's date – calculate the age of that person. To implement the function we will use the `date` type definition given in the script.

```
type date = {day:int,month:int,year:int};

type date = {day:int, month:int, year:int}
```

Using this type definition we can implement the age function as shown below. Note the syntax to access the record fields.

```
fun age(birth:date, today:date) =
    if #year birth > #year today then 0
    else ((#day today + #month today * 30 + #year today * 365)
        - (#day birth + #month birth * 30 + #year birth * 365)) div 365;

val age = fn : date * date -> int
```

To apply the function later on we also define a variable to hold today's date.

```
val today = {day=12,month=11,year=2000};

val today = {day=12,month=11,year=2000} : {day:int, month:int, year:int}
```

Now we are ready to try out our new function using a real life example.

```
age({day=22,month=6,year=1976},today);

val it = 24 : int
```