Computer Science 131, Spring 2002
Assignment 6: Typechecking

Out: Wednesday, February 27
Due: Monday, March 4

Instructions

Important: For this assignment you must use a special version of the SML/NJ compiler; the command on turing is `/cs/cs131/bin/sml-cs131-a6` instead of the usual `sml`.

You should get the file `assign6.sml` (and the test files) from the web page, complete this file, and submit it via `cs131submit` as usual. As always, your file should contain no syntax or type errors.

Introduction

For this assignment, you are to implement a typechecker for our simple functional language, using the ideas discussed in class on Monday. The "concrete syntax" for this assignment is shown in Figure 1. The main changes to be aware of are:

- Pairs are written with parentheses as in ML, instead of angle brackets.
- Anonymous function values must specify the type of their argument
- Anonymous recursive function values (`rec`) require two type annotations: the first gives the type of the argument, and the second gives the type of the result returned by the function. (Careful...neither type is the type of the entire function value!)

The corresponding datatype representation has been predefined for you in `sml-cs131-a6`. This version of the SML compiler includes a structure `A` which implements abstract syntax and two helper functions for printing the abstract syntax; the signature of the structure `A` is shown in Figure 2.

The correspondence between concrete syntax and the datatype representation is shown in Figure 3. There the notation $\lceil exp \rceil$ is used to denote the SML value of type `A.absyn` corresponding to *exp*. However, you don't have to do this translation manually if you don't want to; the the `sml-cs131-a6` includes a function

```
M.parse : string -> A.exp
```

which takes a filename and parses the contents of the file, returning the corresponding abstract syntax. Two sample input files are posted on the course web page; you are strongly encouraged write your own (probably simpler) test cases. Feel free to look at the output of the parser to make sure it's yielding the abstract syntax you're expecting.

1 A Typechecker (90%)

Write a function

```
typeof : A.ty env * A.absyn -> A.ty
```

which, given a typing environment and an expression, returns the expression's type if one exists and raises the `Typeof` exception otherwise. You do *not* have to write an evaluator for this language!

The typechecking rules for function values with type annotations are as follows:

$$\frac{tenv, var_2{:}type_1 \vdash exp : type_2}{tenv \vdash (\texttt{(}var{:}type\texttt{) => } exp) : type_1\texttt{->}type_2}$$

$$\frac{tenv, var_1{:}type_1\texttt{->}type_2, var_2{:}type_1 \vdash exp : type_2}{tenv \vdash (\texttt{rec } var_1\texttt{(}var_2{:}type_1\texttt{)}{:}type_2\texttt{ => } exp) : type_1\texttt{->}type_2}$$

2 Error Messages (10%)

In addition to raising an exception when an error is found, have your typechecker print a helpful error message explaining what the problem was. You should use the built-in function

```
print : string -> unit
```

$$
\begin{array}{rll}
exp ::= & n & \text{(integers)} \\
| & var & \\
| & \texttt{tt} \mid \texttt{ff} & \\
| & \texttt{(}var\texttt{:}type\texttt{)} \texttt{ => } exp & \\
| & \texttt{rec } var_1\texttt{(}var_2\texttt{:}type_1\texttt{):}type_2 \texttt{ => } exp & \\
| & \texttt{(}exp_1\texttt{,}exp_2\texttt{)} & \\
| & exp_1 \texttt{ + } exp_2 \mid exp_1 \texttt{ - } exp_2 \mid exp_1 \texttt{ * } exp_2 & \\
| & exp_1 \texttt{ == } exp_2 \mid exp_1 \texttt{ < } exp_2 \mid exp_1 \texttt{ <= } exp_2 \mid \ldots & \\
| & exp_1 \texttt{ ? } exp_2 \texttt{ : } exp_3 & \\
| & \texttt{let } var \texttt{ be } exp_1 \texttt{ in } exp_2 & \\
| & exp_1\ exp_2 & \\
| & \texttt{fst } exp & \\
| & \texttt{snd } exp & \\
\\
type ::= & \texttt{int} & \\
| & \texttt{bool} & \\
| & type_1\texttt{->}type_2 & \\
| & type_1\texttt{*}type_2 &
\end{array}
$$

Figure 1: Concrete Syntax for this Assignment

```
sig
  (* Representation of types in the Simple Functional Language
   *)
  datatype ty = Int_t
              | Bool_t
              | Arrow_t of ty * ty
              | Times_t of ty * ty

  datatype aop = Plus | Minus | Times | Div
  datatype cop = Less | Equal | LessEq | Greater | GreaterEq | NotEq

  type varname = string

  datatype absyn =
      Num      of int
    | Bool     of bool
    | Var      of varname
    | Arith    of absyn * aop * absyn   (* New *)
    | Compare  of absyn * cop * absyn   (* New *)
    | Cond     of absyn * absyn * absyn
    | Let      of varname * absyn * absyn
    | FnVal    of varname * ty * absyn  (* Modified *)
    | RecFnVal of varname * varname * ty * ty * absyn  (* New *)
    | Apply    of absyn * absyn
    | Pair     of absyn * absyn
    | Fst      of absyn
    | Snd      of absyn


  val ty_toString  : ty    -> string
  val toString     : absyn -> string
end
```

Figure 2: Signature of the structure `A`

$\lceil$`int`$\rceil$	= `A.Int_t`
$\lceil$`bool`$\rceil$	= `A.Bool_t`
$\lceil type_1$`->`$type_2\rceil$	= `A.Arrow_t(`$\lceil type_1\rceil$`,`$\lceil type_2\rceil$`)`
$\lceil type_1$`*`$type_2\rceil$	= `A.Times_t(`$\lceil type_1\rceil$`,`$\lceil type_2\rceil$`)`
$\lceil n\rceil$	= `A.Num(`n`)`
$\lceil$`tt`$\rceil$	= `A.Bool(true)`
$\lceil$`ff`$\rceil$	= `A.Bool(false)`
$\lceil var\rceil$	= `A.Var("`var`")`
$\lceil exp_1$ `+` $exp_2\rceil$	= `A.Arith(`$\lceil exp_1\rceil$`,A.Plus,`$\lceil exp_2\rceil$`)`
$\lceil exp_1$ `-` $exp_2\rceil$	= `A.Arith(`$\lceil exp_1\rceil$`,A.Minus,`$\lceil exp_2\rceil$`)`
$\lceil exp_1$ `*` $exp_2\rceil$	= `A.Arith(`$\lceil exp_1\rceil$`,A.Times,`$\lceil exp_2\rceil$`)`
$\lceil exp_1$ `<` $exp_2\rceil$	= `A.Compare(`$\lceil exp_1\rceil$`,A.Less,`$\lceil exp_2\rceil$`)`
$\lceil exp_1$ `<=` $exp_2\rceil$	= `A.Compare(`$\lceil exp_1\rceil$`,A.LessEq,`$\lceil exp_2\rceil$`)`
$\lceil exp_1$ `==` $exp_2\rceil$	= `A.Compare(`$\lceil exp_1\rceil$`,A.Equal,`$\lceil exp_2\rceil$`)`
$\lceil exp_1$ `?` exp_2 `:` $exp_3\rceil$	= `A.Cond(`$\lceil exp_1\rceil$`,`$\lceil exp_2\rceil$`,`$\lceil exp_3\rceil$`)`
$\lceil$`let` var `be` exp_1 `in` $exp_2\rceil$	= `A.Let("`var`",`$\lceil type\rceil$`,`$\lceil exp\rceil$`)`
$\lceil$`(`var`:`$type$`) =>` $exp\rceil$	= `A.FnVal("`var`",`$\lceil type\rceil$`,`$\lceil exp\rceil$`)`
$\lceil$`rec` var_1`(`var_2`:`$type_1$`):`$type_2$ `=>` $exp\rceil$	= `A.RecFnVal("`var_1`","`var_2`",`$\lceil type_1\rceil$`,`$\lceil type_2\rceil$`,`$\lceil exp\rceil$`)`
$\lceil exp_1\ exp_2\rceil$	= `A.Apply(`$\lceil exp_1\rceil$`,`$\lceil exp_2\rceil$`)`
$\lceil$`(`exp_1`,`exp_2`)`$\rceil$	= `A.Pair(`$\lceil exp_1\rceil$`,`$\lceil exp_2\rceil$`)`
$\lceil$`fst` $exp\rceil$	= `A.Fst(`$\lceil exp\rceil$`)`
$\lceil$`snd` $exp\rceil$	= `A.Snd(`$\lceil exp\rceil$`)`

Figure 3: Datatype Representation of Syntax