CS 510 Programming Languages
Assignment #2

Out: Monday, 23 September 2002
Due: Monday, 7 October 2002 (11:00 am)

A Theory of DeBruijn Indices

DeBruijn indices are a way of representing bound variables conveniently and efficiently. Instead of using variable names (strings), one uses integer indices. DeBruijn indices uniquely identify the binding occurrence of a variable. In general, the index $\boxed{\text{n}}$ refers to the value bound by the nth binding expression counting outward from the occurrence. For example, the two bindings for variable `x` below are clearly distinguished in the DeBruijn representation without a need for a special "shadowing" rule.

Variables	DeBruijn	Value
`let x = 0 in (let x = 3 in x + x)`	`let 0 in (let 3 in` $\boxed{1}$ `+` $\boxed{1}$`)`	6
`let y = 0 in (let x = 3 in x + y)`	`let 0 in (let 3 in` $\boxed{1}$ `+` $\boxed{2}$`)`	3

For these questions we will use the language of arithmetic expressions presented in the H.O.A.S. lecture notes. In the programming part of the assignment, you will be asked to extend this representation technique to the MinML language.

$$\text{Expression} \quad e \;::=\; \boxed{n} \mid \texttt{number}(n) \mid \texttt{plus}(e_1, e_2) \mid \texttt{times}(e_1, e_2) \mid \texttt{let}\,(e_1\,\texttt{in}\,e_2)$$

1. (**Definition of Substitution**) Following the definition of substitution presented in the lecture notes, define substitution ($[e'/\boxed{i}]e$) for arithmetic expressions in DeBruijn form. Both e', the term to be substituted, and e may be open, *i.e.* they may contain free indices. Your definition of substitution must avoid variable capture (when free variables in (e') become bound). Pay particular attention to binding expressions *i.e.* `let`.

   The DeBruijn representation has the characteristic that the same variable may be referred to by different indices at different levels of nesting (see lecture notes for an example). Therefore, you will find it necessary to define an auxiliary function $\uparrow_k(e)$ (pronounced shift) which operates on expressions. This function should increment the "free" DeBruijn indices in an expression. The intention is that free and bound variables will be distinguished by the subscript k; free variables are those indices i such that $i \geq k$.

   Your definitions for substitution and the shift operator should have the following form:

$$[e'/\boxed{i}]e_1 \quad = \quad e_2$$

$$\uparrow_k(e_1) \quad = \quad e_2$$

2. (**Translation to DeBruijn form**) Define a translation to DeBruijn form $\mathtt{Tr}(X, e)$ for closed arithmetic expressions. The definition will be structured by induction on the expression e. The key idea is to store bound variable names in an environment list, so that the nesting level of their binding expression can be recalled. Formally, an environment is characterized by:

$$\begin{array}{rrcl} \text{Environment} & X & ::= & nil \mid X; x \\ \text{Variable} & x & ::= & string \end{array}$$

   An auxiliary "lookup" function will also be necessary, defined by induction on the environment X. The lookup function should compute the position of a variable in the environment list.

3. (**Weakening Lemma**) Prove the following theorem for your definition of shift and translation:

$$x \notin FV(e) \quad \Rightarrow \quad \mathtt{Tr}(X; x; y_k; y_{k-1}; ...; y_1, e) \;= \uparrow_k (\mathtt{Tr}(X; y_k; y_{k-1}; ...; y_1, e))$$

   This lemma is called a "weakening" lemma because you are removing the unnecessary entry x from $X; x; y_k; y_{k-1}; ...; y_1$, thereby weakening the environment. An environment with fewer entries is considered to be weaker because it contains less information about bound variables than the original environment.

4. (**Commutativity of Translation/Substitution**) As a sanity check, we should verify that the definition of substitution on DeBruijn terms is consistent with the definition of substitution on expressions with named variables. Prove for your definition of translation and substitution:

$$\mathtt{Tr}(X, [e'/x]e) \quad = \quad [\mathtt{Tr}(X, e')/\mathtt{Tr}(X, x)]\mathtt{Tr}(X, e)$$

   You may assume that X contains entries for all the free variables of e, in particular it contains an entry for x. This is, in fact, the case when translating programs consisting of closed expressions e, starting with the environment $X = nil$. During the course of the proof, you should find the "weakening" lemma helpful.

B Programming Work

For the programming part of this assignment you will implement a parser, typechecker, and evaluator for MinML. In the assignment folder you'll find several files with support code; you will only need to fill in the missing code in `parse.sml`, `eval.sml`, and `typing.sml` to complete the assignment.

As a general rule, I've left blank lines in these files equal to the number of lines in the solution. You can expect to sometimes come up with a shorter or longer solution, but if you find yourself writing code that's five or ten times the length of ours, you probably want to reconsider your strategy.

Before you begin, you may wish to read through the provided code (especially the signatures) to gain an understanding of the setup. All of the necessary SML files are listed in the `sources.cm` file, and you can build the project in SML/NJ by typing `CM.make()`.

B.1 Parser

The file `parse.sml` contains the beginnings of a parser for MinML. The `parse` function turns a `Lex.token Stream.stream` into a `MinML.exp Stream.stream` by consuming programs (which are expressions followed by a semicolon). For simplicity, we will not attempt error recovery; if your parser encounters an error it should just raise the exception ParseError with an informative message.

Here is the concrete syntax we will use. To simplify your job further, this syntax uses no infix notation.

```
Op ::= PLUS | MINUS | TIMES | EQUALS | NEGATE

Type ::= INT | BOOL | ARROW Type Type

ExpSeq ::= Exp | Exp COMMA ExpSeq

Var ::= VAR(s)

Exp ::= LPAREN Exp RPAREN
      | NUMBER(n)
      | Var
      | Op LPAREN ExpSeq RPAREN
      | Exp Exp
      | TRUE
      | FALSE
      | IF Exp THEN Exp ELSE Exp FI
      | FUN Var LPAREN Var COLON Type RPAREN COLON Type IS Exp END

Program ::= Exp SEMICOLON
```

We provide one critical but tedious part of parsing: the lexer. The lexer takes a raw character stream and creates a stream of tokens from it. The BNF diagram above makes reference to these tokens; they are listed in the table below along with the textual symbol which produces them.

Symbol	**Lexer.token**
`int`	`INT`
`bool`	`BOOL`
`->`	`ARROW`
`true`	`TRUE`
`false`	`FALSE`
`fun`	`FUN`
`is`	`IS`
`end`	`END`
`if`	`IF`
`then`	`THEN`
`else`	`ELSE`
`fi`	`FI`
`,`	`COMMA`
`(`	`LPAREN`
`)`	`RPAREN`
`;`	`SEMICOLON`
`~`	`NEGATE`
`=`	`EQUALS`
`*`	`TIMES`
`-`	`MINUS`
`+`	`PLUS`
`:`	`COLON`
n	`NUMBER(n)`
any other string s	`VAR(s)`

This syntax should be mostly self-explanatory. Note in particular that primitive operations are prefix (followed by one or more expressions separated by commas and surrounded by parentheses). Types are also written in prefix notation. Application is written by juxtaposition, as in ML. Here are some examples along with their translation into MinML abstract syntax (MinML.exp).

Concrete Syntax	**Lexer Tokens**	**Abstract Syntax**
`true`	`TRUE`	`Bool(true)`
`1`	`NUMBER(1)`	`Int(1)`
`if true then 4 else 5 fi`	`IF TRUE THEN NUMBER(4) ELSE NUMBER(5) FI`	`If(Bool(true), Int(4), Int(5))`
`f 3`	`VAR(‘‘f’’) NUMBER(3)`	`Apply(Var(‘‘f’’), Int(3))`
`+(1, 2)`	`PLUS LPAREN NUMBER(1) COMMA NUMBER(2) RPAREN`	`Primop(Plus, Int(1), Int(2))`
`fun f(g : -> int -> bool int) : bool is true end`	`FUN VAR(‘‘f’’) LPAREN VAR(‘‘g’’) COLON ARROW INT ARROW BOOL INT RPAREN COLON BOOL IS TRUE END`	`Fun(‘‘f’’, ‘‘g’’, ARROW(INT,ARROW(bool,int)), BOOL, Bool(true))`

For human convenience, we choose to parse programs in the named variable syntax. However, these will be translated to DeBruijn expressions immediately after parsing. **All further implementation work** on the typechecker and evaluator **will be based on DeBruijn expressions**, not the named variable form.

You can test your parser before you complete the typechecker and evaluator. Run `Top.loop_print ()` or `Top.file_print "test_file"`.

Task: Translation to DeBruijn Form

In file `parse.sml`, complete the implementation of function `Parse.translate`. When completed, it should translate closed MinML expressions in the named variable representation (type MinML.exp) to closed expressions in the DeBruijn representation (type DBMinML.exp).

Your work on arithmetic expressions in the theory section should make this easier, but you will need to extend the translation to deal with primitive operations, `fun` expressions, and other syntactic elements specific to MinML. In particular, MinML's `fun` expressions bind *two variables* (to the function and argument, respectively) rather than one variable, as was the case of `let`. We choose to adopt the convention that $\boxed{1}$ refers to the argument and $\boxed{2}$ to the function itself. For example:

Named	**DeBruijn**

`fun fact (x : int) : int is`
`if =(x, 0) then 1 else *(x,fact(-(x,1)))`
`fi`
`end`
	`fun _ (_ : int) : int is`
	`if =(`$\boxed{1}$`, 0) then 1 else *(`$\boxed{1}$`,`$\boxed{2}$`(-(`$\boxed{1}$`,1)))`
	`fi`
	`end`

B.2 Typechecker

Next, we ask you to implement a typechecker for MinML. The static semantics for MinML, given in Figure 1, assures that every expression has at most one type. Therefore, your typechecker will return the unique type for an expression if it is well-typed, or raise the exception `TypeError` otherwise.

Recall that the specification of MinML uses a *typing judgment* to classify MinML expressions as ill- or well-typed. The typing judgment is defined inductively by the inference rules. Therefore, in order to decide

$$\frac{}{\Gamma; \tau \vdash \boxed{1} : \tau} \; VarTyp1$$

$$\frac{\Gamma \vdash \boxed{k} : \tau}{\Gamma; \tau' \vdash \boxed{k+1} : \tau} \; VarTypK$$

$$\frac{}{\Gamma \vdash n : \texttt{int}} \; NumTyp$$

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{bool}} \; TrueTyp$$

$$\frac{}{\Gamma \vdash \texttt{false} : \texttt{bool}} \; FalseTyp$$

$$\frac{\Gamma \vdash e_1 : \tau_2 \to \tau \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 \; e_2 : \tau} \; AppTyp$$

$$\frac{\Gamma \vdash e_c : \texttt{bool} \quad \Gamma \vdash e_t : \tau \quad \Gamma \vdash e_f : \tau}{\Gamma \vdash \texttt{if}\, e_c\, \texttt{then}\, e_t\, \texttt{else}\, e_f\, \texttt{fi} : \tau} \; IfTyp$$

$$\frac{\Gamma \vdash e_1 : \tau_{o1} \quad \ldots \quad \Gamma \vdash e_n : \tau_{on}}{\Gamma \vdash o(e_1, \ldots, e_n) : \tau_o} \; OpTyp$$

$$\frac{\Gamma; (\tau_1 \to \tau_2); \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{fun}\, f(x : \tau_1) : \tau_2\, \texttt{is}\, e\, \texttt{end} : \tau_1 \to \tau_2} \; FunTyp$$

Figure 1: Static Semantics for MinML

whether a given expression has a type, we need to search for a derivation using the typing rules. However, a moment of thought realizes that if we could classify an expression as ill- or well-typed through a typing judgment, then we could also retrieve its type easily, because the derivation itself would tell exactly how to determine the type of the expression. Therefore we see that deciding the type of an expression is no harder than simply classifying the expression as ill- or well-typed.

In general, we cannot assume that an expression matches only one typing rule and thus, the search strategy for a derivation can be *non-deterministic*. Fortunately, the search strategy for MinML is *syntax directed*: the form of expression we are typing determines uniquely which rule to apply. Therefore, if the typechecker finds that no rule can be applied to an expression, it knows that the expression is ill-typed and can raise an exception immediately without backtracking. Your code will probably have one function clause for each clause of the datatype `DBMinML.exp`.

We provide in `minml.sml` a function `typeOfPrimop` which returns the domain and range types for a primop. Your typechecker should make use of this function, but should not rely on the fact that our primops currently have a maximum of two arguments (we should be able to extend MinML with additional primops without modifying your type checker).

Task: Typechecker

Complete the code in `typing.sml` to produce a structure `Typing :> TYPING` which implements the behavior specified. You should not modify any other files. Remember that the expression to be typechecked will be in DeBruijn form. This file contains some code to get you started; we recommend using it. The majority of your work will be done on the function `typing`, with type `(typ env) * exp -> typ` and specification as follows:

Given a type environment Γ and an expression e, `typing` should return τ, the type of e under Γ, if τ exists. If e is ill-typed in Γ then `typing` should raise the exception `TypeError`.

You can test your typechecker before you complete the evaluator. Run `Top.loop_type ()` or `Top.file_type "test_file"`.

B.3 Evaluator

Finally, you'll implement the MinML dynamic semantics in the file `eval.sml`. The dynamic semantics are given (in figure 2) as a relation "$\mapsto$" for single-step evaluation. There are many more efficient ways of evaluating MinML programs (as we'll see later in the class), but we require that you strictly follow the specified semantics for this assignment.

The evaluation algorithm is straightforward. First it will use the "search rules" *OpArg*, *IfCond*, *AppFun*, and *AppArg* to recursively scan the input expression for the proper subexpression to modify. Once the proper subexpression has been located, one of the "instruction rules" *OpVals*, *IfTrue*, *IfFalse*, *CallFun* can be applied. If none of the rules apply (as might happen if the expression is already fully evaluated, or is ill-typed), the evaluator will raise an exception.

For example, on the expression $e_1\ e_2$, the evaluator will try to apply an evaluation step to the function expression e_1. If it is already a value, the evaluator will try to apply a step to the argument expression e_2. If it is already a value as well, the evaluator will try to use the instruction rule for application, *CallFun*.

Since the *CallFun* rule needs to substitute the argument (and the function itself) into the body of the function, you will also need to implement a substitution algorithm. Use the definition of substitution on arithmetic expressions in DeBruijn form that you derived in the theory section as a guide. The definition will of course need to be extended to handle MinML expressions and their quirks (such as the fact that `fun` binds two variables).

Task: Evaluator

Complete the code in `eval.sml` to produce a structure `Eval :> EVAL` which implements the behavior specified. You should not modify any other files. Some of this code has been written for you. Most of the work that you do will be in the function `step`, which has type `exp -> exp` and specification as follows:

Given an expression e in DeBruijn form, `step` returns the unique e' where $e \mapsto e'$. If such an e' does not exist, `step` raises the exception `Stuck`.

We've provided a few test cases. Once your assignment is complete, these may be run in the following manner:

```
Top.file_eval "test_file.mml";
```

Filename	**Expected Result**	**Description**
`if.mml`	`3 : int`	Simple test of `if`
`fun.mml`	`fun ident(x : int) : int is x end : -> int int`	Simple test of `fun`
`factorial.mml`	`120 : int`	The factorial function
`self.mml`	*ill-typed*	Ill-typed function
`hof.mml`	`7 : int`	Simulates pairs using functions

These test files are not exhaustive, so you should develop your own in order to test your program thoroughly. Remember, however, that working on test cases is only a *necessary* condition for getting a good grade on your homework. The elegance of your solution is important and will be taken into account when grading.

C Hand-in Instructions

You must turn in the following items.

Component Name	**Contents**
`assn1`	Your answers to part A
`parse.sml`	a structure `Parse :> PARSE`
`typing.sml`	a structure `Typing :> TYPING`
`eval.sml`	a structure `Eval :> EVAL`
`examples`	a few well-chosen examples of your code

$$\frac{}{o(v_1, \ldots, v_n) \mapsto v}\ OpVals$$

$$\frac{}{\texttt{if true then } e_t \,\texttt{else}\, e_f \,\texttt{fi} \mapsto e_t}\ IfTrue$$

$$\frac{}{\texttt{if false then } e_t \,\texttt{else}\, e_f \,\texttt{fi} \mapsto e_f}\ IfFalse$$

$$\frac{v_2 = \texttt{fun}\,\_(\_ : \tau_1) : \tau_2 \;\texttt{is}\; e \;\texttt{end}}{v_2\ v_1 \mapsto [\,v_2/\boxed{2}\,][\,v_1/\boxed{1}\,]e}\ CallFun$$

$$\frac{e_i \mapsto e_i'}{o(v_1, \ldots, e_i, \ldots, e_n) \mapsto o(v_1, \ldots, e_i', \ldots, e_n)}\ OpArg$$

$$\frac{e_c \mapsto e_c'}{\texttt{if } e_c \texttt{ then } e_t \texttt{ else } e_f \texttt{ fi} \mapsto \texttt{if } e_c' \texttt{ then } e_t \texttt{ else } e_f \texttt{ fi}}\ IfCond$$

$$\frac{e_1 \mapsto e_1'}{e_1\ e_2 \mapsto e_1'\ e_2}\ AppFun$$

$$\frac{e_2 \mapsto e_2'}{v_1\ e_2 \mapsto v_1\ e_2'}\ AppArg$$

Figure 2: Dynamic Semantics for MinML: v, v_i, etc. denote expressions which are values