Assignment 2:
Implementing MinML

15-312: Foundations of Programming Languages
Kevin Watkins (kw@cmu.edu)

Out: Thursday, 27 Jan 2004
Due: Thursday, 10 February 2004 (11:59 PM)
100 points total

1 Introduction

This is our first programming assignment. For this assignment you will implement a typechecker and evaluator for MinML in SML.

The code for this assignment can be found in

/afs/andrew/scs/cs/15-312/handout/2/

In the assignment folder you’ll find several files with support code; you will only need to fill in the missing code in translate.sml, typing.sml, and eval.sml.

1.1 Handins

This assignment must be turned in electronically by 11:59 pm on Thursday, 10 February 2004. The files translate.sml, typing.sml, and eval.sml must be handed in to

/afs/andrew/scs/cs/15-312/handin/<yourid>/2/

Your code must compile with the original versions of the other files, so don’t change them. If your code does not compile and type-check, you will receive little or no credit. Note that only the three files listed above will actually be copied out of your directory, so don’t do anything like adding more files to sources.cm.

Also, please turn in any test cases you’d like us to use by copying them to your handin directory. To ensure that our scripts notice the files, make sure they have the suffix .mml.

Immediately after the deadline, we will run a script to sweep through all the handin directories and copy your files elsewhere. We will also sweep 24, 48, and 72 hours after the deadline, for anyone using late days on this assignment. It is not a good idea to work on your code in the handin directory, because if you happen to be working when a sweep happens, it might pick up a broken version of your code. Instead, work on your code somewhere else, and copy it into your
handin directory whenever you have a working version. Note that your handin directory may briefly be write-protected while a sweep is actually happening.

You can write into your handin directory as often as you like. The last distinct version picked up by the sweeps will be graded. If you would rather have an earlier version graded (perhaps because you want to save on late days) you must tell Kevin Watkins before the expiration of the three day late handin period.

If you work on your code in AFS, you are responsible for keeping your working code inaccessible to other students in the class using AFS permissions. If you have world-readable code for an assignment in your AFS directory and some other student ends up submitting code identical to yours, you will have no way of defending yourself against charges of unacceptable collaboration.

For more information on handing in code, refer to

http://www.cs.cmu.edu/~crary/312/assignments.html

1.2 Grading

Code for this assignment is graded on correctness and clarity. Don’t try to make your code more efficient if it costs clarity, because you are not being graded on efficiency. For your reference, the sample solution to this assignment has the following line counts:

```
  translate.sml   30
  typing.sml     52
  eval.sml       63
```

If you find yourself writing a lot more code than this, you may want to rethink your approach.

2 Overview of the supplied code

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(). The following files are included:

```
minml.sml     Abstract syntax of MinML
stream.sml    Lazy streams
input.sml     Input as a lazy stream of characters
lexer.sml     Lexical analysis
parse.sml     Parsing concrete syntax into abstract syntax
print.sml     Printing abstract syntax as concrete syntax
translate.sig Signature for translation from named to de Bruijn form
translate.sml (To be filled in by you)
typing.sig    Signature for type checker
typing.sml    (To be filled in by you)
eval.sig      Signature for evaluator
eval.sml      (To be filled in by you)
loop.sml      Combinators for the top-level loop
top.sml       The top-level loop
```
\[ k ::= 0 \mid 1 \mid ... \]
\[ o ::= + \mid - \mid \ast \mid \text{=} \mid < \mid \neg \]
\[ \tau ::= \text{int} \mid \text{bool} \mid \tau_1 \rightarrow \tau_2 \]
\[ e ::= \text{num}(k) \mid \text{true} \mid \text{false} \mid x \mid \text{fun}(\tau_1, \tau_2, f . x . e) \]
\[ \mid o(e_1, \ldots, e_n) \mid \text{if}(e_1, e_2, e_3) \mid \text{apply}(e_1, e_2) \mid \text{let}(e_1, x . e_2) \]

Figure 1: Abstract syntax for MinML

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Token</th>
<th>Symbol</th>
<th>Token</th>
<th>Symbol</th>
<th>Token</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>INT</td>
<td>bool</td>
<td>BOOL</td>
<td>(\rightarrow)</td>
<td>ARROW</td>
</tr>
<tr>
<td>true</td>
<td>TRUE</td>
<td>false</td>
<td>FALSE</td>
<td>fun</td>
<td>FUN</td>
</tr>
<tr>
<td>is</td>
<td>IS</td>
<td>end</td>
<td>END</td>
<td>if</td>
<td>IF</td>
</tr>
<tr>
<td>then</td>
<td>THEN</td>
<td>else</td>
<td>ELSE</td>
<td>let</td>
<td>LET</td>
</tr>
<tr>
<td>in</td>
<td>IN</td>
<td></td>
<td></td>
<td>(</td>
<td>LPAREN</td>
</tr>
<tr>
<td>;</td>
<td>SEMICOLON</td>
<td>~</td>
<td>NEGATE</td>
<td>=</td>
<td>EQUALS</td>
</tr>
<tr>
<td>&lt;</td>
<td>LESSTHAN</td>
<td>*</td>
<td>TIMES</td>
<td>\dash</td>
<td>MINUS</td>
</tr>
<tr>
<td>+</td>
<td>PLUS</td>
<td>:</td>
<td>COLON</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(k)</td>
<td>NUMBER((k))</td>
<td>(x)</td>
<td>VAR((x))</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 2: MinML tokens

2.1 Abstract syntax

The named-style abstract syntax of MinML is shown in Figure 1. For this assignment we will work with both named and de Bruijn-form abstract syntax. In minml.sml there are signatures and structures for each. The primitive operators of MinML such as + or < are all represented by a single SML constructor \(\text{Primop}\). Code in minml.sml indicates the types of each primitive operator and how they evaluate. The abstract syntax groups binders with their scopes, in higher-order abstract syntax style.

2.2 Parsing

The parser provided to you is based on lazy memoizing streams, which should be familiar from 15-212. The code implementing streams is in stream.sml. Parsing itself happens in three stages.

- First, a lazy stream of characters is constructed. Forcing this stream causes the user to be prompted for input, or characters to be read from a file, depending on what the current input source is. This input code is in input.sml.

- Next, the stream of characters is transformed into a stream of tokens by a lexical analyzer in lexer.sml. The concrete syntax of tokens is shown in Figure 2. Each sort of token has a
program ::= exp0 ;
exp0 ::= exp0 = exp1 | exp0 < exp1 | exp1
exp1 ::= exp1 + exp2 | exp1 - exp2 | exp2
exp2 ::= exp2 * exp3 | exp3
exp3 ::= exp∞ | exp4
exp4 ::= exp4 exp∞ | exp∞
exp∞ ::= (exp0) | k | x | true | false
| if exp0 then exp0 else exp0 end
| let x = exp0 in exp0 end
| fun f (x : type0) : type0 is exp0 end
type0 ::= type∞ -> type0 | type∞
type∞ ::= (type0) | int | bool

Figure 3: MinML concrete syntax

• Finally, the stream of tokens is transformed into a stream of expressions by the parser in parse.sml. The parser accepts the concrete grammar shown in Figure 3. Note that the expressions produced by the parser are in named form MinML.exp, not de Bruijn form DBMinML.exp.

The grammar should be mostly self-explanatory. Primitive operations have the same precedences as in SML, and are left-associative. Application is shown by writing the function $e_1$ and the argument $e_2$ next to each other as in SML: $e_1 e_2$. Function types are written $\tau_1 \rightarrow \tau_2$, where $\rightarrow$ is right-associative, as in SML. Some examples of MinML syntax at each of the three stages of parsing are shown in Figure 2.2.

2.3 Printing

Code for printing both named- and de Bruijn-style abstract syntax is provided for you in print.sml. The printer tries to eliminate redundant parentheses where possible.

2.4 Top-level loop

The files loop.sml and top.sml implement a top-level loop for MinML similar to SML/NJ’s. The top-level loop reads expressions one at a time, and performs some action on each one. Using the combinators in loop.sml you can set up a top-level loop that performs actions like type-checking and stepwise evaluation. Some useful top-level loops are already defined in top.sml.

2.5 Test cases

We’ve also provided a few test cases in files of the form foo.mml.
Concrete syntax | Lexer tokens | Abstract syntax
---|---|---
true | TRUE | Bool(true)
1 | NUMBER(1) | Int(1)
if true then 4 else 5 end | IF TRUE THEN NUMBER(4) ELSE NUMBER(5) END | If(Bool(true), Int(4), Int(5))
f 3 | VAR("f") NUMBER(3) | Apply(Var("f"), Int(3))
1 + 2 | NUMBER(1) PLUS | Primop(Plus, [Int(1), Int(2)])
1 + g 2 * 3 | NUMBER(1) PLUS VAR("g") NUMBER(2) TIMES NUMBER(3) | Primop(Plus, [Int(1), Primop(Times, [Apply(Var("g"), Int(2)), Int(3)])])
fun f(g : int -> bool -> int) : bool is true end | FUN VAR("f") LPAREN VAR("g") COLON INT ARROW BOOL ARROW INT RPAREN COLON BOOL IS TRUE END | Fun(ARROW(INT, ARROW BOOL, INT)), BOOL, ("f", "g", Bool(true)))

Figure 4: Examples of MinML parsing

<table>
<thead>
<tr>
<th>Filename</th>
<th>Expected Result</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>if.mml</td>
<td>3 : int</td>
<td>Simple test of if</td>
</tr>
<tr>
<td>fun.mml</td>
<td>fun ident(x : int) : int is x end : int -&gt; int</td>
<td>Simple test of fun</td>
</tr>
<tr>
<td>factorial.mml</td>
<td>120 : int</td>
<td>The factorial function</td>
</tr>
<tr>
<td>self.mml</td>
<td>ill-typed</td>
<td>Ill-typed function</td>
</tr>
<tr>
<td>hof.mml</td>
<td>7 : int</td>
<td>Simulates pairs using functions</td>
</tr>
</tbody>
</table>

These can be run using the functions `Top.file something` in `top.sml`. These test files are (obviously) not exhaustive, so part of this assignment is developing your own test files in order to test each case in your code thoroughly. Remember, however, that passing test cases only helps you make your code correct; you will also be graded on clarity.

You are encouraged to submit your test cases to us. After everyone’s homework is submitted,
we will collect all the test cases together with some of our own, and test your code with all of them. The results of these tests will be factored into the correctness part of your grade. So, although you will not receive extra points directly for submitting test cases, it is still in your interest to make up some that your code handles correctly, since they will tend to improve your grade. See the beginning of the assignment for instructions on how to submit your test cases.

To play around with the parser and become familiar with MinML, type

Top.loop_print_noDB ();

or

Top.file_print_noDB "test_file.mml";

These will print the program (with some redundant parentheses) in the named-variable form.

3 Your tasks

3.1 Variables and binding

In this part of the assignment, you will translate MinML terms from the named representation to a de Bruijn representation. To get started, read Section 5.4 of Robert Harper’s notes, then think about how to translate the abstract syntax. Most cases are very straightforward; variables, let, and fun require a little thought. For fun, which binds two variables at once (the function and its argument), use the convention that the de Bruijn index #1 refers to the argument and the de Bruijn index #2 to the function itself. For example:

<table>
<thead>
<tr>
<th>Named</th>
<th>de Bruijn</th>
</tr>
</thead>
<tbody>
<tr>
<td>fun fact (x : int) : int is if x=0 then 1 else x*fact(x-1) end</td>
<td>fun λ (λ : int) : int is if #1=0 then 1 else #1*#2(#1-1) end</td>
</tr>
</tbody>
</table>

Task: Translation to de Bruijn form (20 points)

In file translate.sml, complete the implementation of function Translate.translate. When completed, it should translate a closed MinML expression in the named variable representation (type MinML.exp) to a closed expression in the de Bruijn representation (type DBMinML.exp). Your implementation of Translate.translate should satisfy the following specification:

If \( e \) is a closed named-form term, then \( \text{Translate.translate}(e) = e' \), where \( e' \) is its de Bruijn-form counterpart.

If \( e \) is not closed, then \( \text{Translate.translate}(e) = \text{raise Translate.Error}(s) \) for some informative string \( s \).

In the translator, you will need to maintain an environment of variable names. Use the simplest representation possible; don’t worry about efficiency.
3.2 Static semantics

Next, you will implement a typechecker for MinML in de Bruijn form. The static semantics for MinML, given in Figure 5, has the property that every expression has at most one type. Your typechecker will return the unique type for an expression if it is well-typed, or raise the exception Typing.Error\( (s) \) with some informative string \( s \) 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 whether a given expression has a type, we need to search for a derivation using the typing rules.

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. Otherwise, the typechecker can apply the (unique) appropriate typing rule. Your code will probably have one function clause for each constructor of the datatype DBMinML.exp.

We provide a function MinML.typeOfPrimop that returns the domain and range types for a primop. These correspond to the types \( \tau_{o1} \) and \( \tau_o \) in the rule OpTyp. Your typechecker should use this function. It should remain correct even if the language were later extended with additional primops (potentially having zero or more than two arguments).
Task: Typechecker (35 points)

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 de Bruijn form. Your function Typing.typeOf should have the following property.

If \( \vdash e : \tau \) for some (unique) \( \tau \), then Typing.typeOf(e) = \( \tau \).

If \( \vdash e : \tau \) does not hold for any \( \tau \), then Typing.typeOf(e) = raise Typing.Error(s) for some \( s \).

Be sure that you implement this specification exactly.

You can test your typechecker at this point before going on. Run Top.loop_type () or Top.file_type "test_file.mml".

3.3 Dynamic semantics

Finally, you’ll implement the MinML dynamic semantics in the file eval.sml. The dynamic semantics is given in Figure 6 as a relation “\( \mapsto \rightarrow \)" 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. Before reading the figure, you should decide what MinML expressions should be values.

The evaluation algorithm is straightforward. First it will use the "compatibility rules" OpArg, IfCond, AppFun, AppArg, and LetArg to recursively scan the input expression for the proper subexpression to modify. Once the proper subexpression has been located, one of the “reduction rules” OpVals, IfTrue, IfFalse, CallFun, Let can be applied. If no rule applies (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 reduction rule for application, CallFun.

You should use the function MinML.evalPrimop to compute the result of the rule OpVals.

Since the CallFun and Let rules involve substitutions, you will also need to properly implement substitutions. You may find your notes from the recitation on de Bruijn indices helpful for this part.

Task: Evaluator (45 points)

In eval.sml, fill in the structure Eval :> EVAL to implement the behavior specified. You should not modify any other files. You should implement the function Eval.isValue according to the following specification:

Suppose \( e \) is a closed de Bruijn term.

If \( e \) is a value, then Eval.isValue(e) = true.

If \( e \) is not a value, then Eval.isValue(e) = false.

If \( e \) is not closed, then Eval.isValue(e) can do anything.
You should implement the function `Eval.step` according to the following specification:

Suppose \( e \) is a closed de Bruijn term.

If \( e \mapsto e' \) for some (unique) \( e' \), then `Eval.step(e) = e'`.

If \( e \mapsto e' \) does not hold for any \( e' \), then `Eval.step(e) = raise Eval.NoStep`.

If \( e \) is not closed, then `Eval.step(e)` can do anything.

Be careful to implement these specifications exactly according to the definitions of the judgments. It is easy to get some corner cases wrong.

You can test your evaluator by running `Top.loop_eval()` or `Top.file_eval "test_file.mml"`. You can use the functions `Top.loop_step` and `Top.file_step` to see the individual evaluation steps until a value or stuck state is reached.