CS 6371	**Advanced Programming Languages**	Assignment 7:
Spring 2011	Due: Apr 7, 2011 by 1:05pm	Functional TIMP

1 Introduction

In this assignment we will introduce functions to TIMP. Functions in TIMP will be *first-class*, which means that programmers can pass them as parameters to other functions or assign them to variables. We will also allow recursion and mutual recursion. Our language won't include explicit support for currying or partial evaluation the way OCaml does, but programmers will be able to simulate both by writing functions that return functions. We will not support polymorphism in our language.

I will provide you with a lexer and a parser for the new language. Your job is to extend your type-checker solution from assignment 5 so that it also type-checks function definitions and function calls. (If your solution from assignment 5 did not work, you may start with the instructor-provided solution to assignment 5 and extend that instead.) You must also extend the evaluator from assignment 5 so that it supports functions.

2 Support Code

2.1 Overview of Files

Obtain the starting code for the assignment by downloading the following four text files from eLearning (click on the link for Assignment 7):

```
imptypes.ml
impparser.mly
implexer.mll
imp.ml
```

Please note that the files for this assignment are *different* from the files that were provided for assignments 2 and 5; you must download the new files in order to complete the assignment.

The `imptypes.ml` file defines the OCaml data structures that we will use to model TIMP programs with functions. You should read through the file to familiarize yourself with the types defined therein. Note that the `ityp` type has been extended with a new TIMP type for functions. A TIMP function has type `TypFunc(tl,rt)` where `tl` is a list of the types of the function's arguments and `rt` is the type of the function's return value.

In addition, the `iexpr` type has been extended with two new constructors—one for function abstraction and one for function application. An `Abstraction(x,c,_)` expression is an anonymous function similar to $(\lambda x.c)$ in the λ-calculus or `(fun x -> c)` in OCaml. Here, `x` is a LIST of the function's parameters and their types. (Unlike the λ-calculus, our language will have built-in support for functions with multiple arguments.) Parameter `c` is the TIMP

command that constitutes the body of the function. An `Apply(e,el,_)` expression applies the anonymous function given by expression `e` to the list of arguments given in expression list `el`.

Like assignments 2 and 5, you do not need to look at the `impparser.mly` or `implexer.mll` files to complete this assignment. They tell OCaml how to turn a text file into an `icmd` value.

The `imp.ml` file is where all of your code should go. (Please do not modify any of the other files!) The top of the file is unchanged from assignment 5, but if you scroll down to the interpreter section you will see some differences: In assignment 5, stores were mappings from variables to integers; in this assignment they also map variables to code. Thus, a `store` has type `varname` $\rightarrow$ `heapval`, where a `heapval` is either an integer (`Data n`) or code (`Code (p,c)`). Code consists of a list `p` of the variable names of the function's formal parameters and a command `c` comprising the function body. Command `c` returns a value by assigning to a variable named `ret`. The value of `ret` when `c` finishes executing is the return value of the function.

To complete the assignment, you need to implement two functions:

1. Implement a `typchk_expr` function that type-checks an expression e given a typing context tc. Start by copying your assignment 5 solution for `typchk_expr` (or use the sample solution for assignment 5 if yours was incorrect). Extend this implementation by adding two new cases—one for `Abstraction` and one for `Apply`. Your type-checker should implement typing rules 36 and 37 from §5.3.

2. Finish the implementation of `eval_expr` by filling in the cases for `Abstraction` and `Apply`. Your implementation should implement operational semantics rules 20 and 21 provided in §5.2. If the interpreter ever encounters a stuck state (e.g., it is asked to "apply" an integer as if it were a function), it may throw the `SegFault` exception or take any other implementation-defined action. (Your type-checker should reject any program capable of getting stuck, so this should never actually happen in a correct implementation!)

Hint: There are several `List` library functions that might be helpful as you manipulate lists of function arguments. In my solution I used `List.combine`, `List.filter`, `List.fold_left`, `List.length`, and `List.map`. Each of these are documented in the online OCaml manual.

2.2 Build Procedure

The build procedure is the same as for assignments 2 and 5. The instructions are repeated below for your reference.

Once you acquire the four starting files, put them all in the same directory somewhere, and make an initial build of the software by executing the following commands in this order:

```
ocamlc -c imptypes.ml
ocamlyacc impparser.mly
ocamlc -c impparser.mli
ocamlc -c impparser.ml
ocamllex implexer.mll
ocamlc -c implexer.ml
ocamlc -c imp.ml
ocamlc -o imp.exe imptypes.cmo implexer.cmo impparser.cmo imp.cmo
```

(When compiling on Unix, omit the `.exe` extension from `imp.exe` in the final command.) The OCaml programs `ocamlc`, `ocamlyacc`, and `ocamllex` must all be in your path in order for this to work. (They are all located in the same directory, so if `ocamlc` is in your path, the other two should be also.) The first six of these commands should only need to be executed once. As you edit `imp.ml`, you should only need to execute the last two commands to recompile your new code.

2.3 Testing

Two sample TIMP programs are provided with which you can test your solution. These are the same two programs that were provided for assignment 5, except that they've been modified to use functions.

- `factorial.imp` computes the factorial of its input. To test it, first build the interpreter (see section 2.2) and then execute:

  ```
  imp factorial.imp 5
  ```

  If your evaluator code is correct, then the program should print 120 (= 5!).

- `isprime.imp` yields `true` if its input is a prime number and `false` otherwise. To test it, execute:

  ```
  imp isprime.imp 100
  ```

  If your evaluator code is correct, then the program should print `false` (because 100 is not a prime number).

Both of the sample programs are well-typed, so your type-checker should not signal a typing-error for them. To fully test your type-checker implementation you will want to write some TIMP programs with functions that are not well-typed.

3 Submission of Solutions

You should submit your solution to this assignment through eLearning as a single file named `lastname.ml` (where `lastname` is your last name). As always, please put a comment at the top of your file that includes your name and email address. You may modify anything in `imp.ml`, but do not modify any of the other files without permission from the instructor. To test your solutions, I will compile them using the same steps as described in section 2.2.

4 Grading

Your assignments will be graded on a scale of 0–10, where each grade is assigned as follows:

10: *a perfect solution; elegant code and no bugs found*
9: *somewhat inelegant, but it seems to work*
7–8: *mostly correct but one or two small bugs*
5–6: *major flaws for one or two cases*
2–4: *solution compiles, but significant code missing*
1: *could not get your solution to compile*
0: *no submission / late submission*

Although the grading is out of only 10 points, all homeworks in the course will be weighted equally when computing your final grade.

Commenting your code can help earn you partial credit. In particular, if you cannot figure out how to solve part of the problem, include comments that describe what you were trying to do and why you couldn't get it to work. Be sure that your code compiles even if you know it doesn't correctly type-check some TIMP programs, since code that compiles will earn you a grade of at least 2. If you know that your code fails on some inputs, you should say so in your comments. This can sometimes make the difference between grades of 2, 3, and 4, since knowing that there is a bug means you were closer to a correct solution than if you didn't realize there was an error.

5 Reference

5.1 Syntax of TIMP

commands	$c ::= \texttt{skip} \mid c_1; c_2 \mid v\texttt{:=}e \mid \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2$ $\mid \texttt{while}\ e\ \texttt{do}\ c \mid td\ v$
expressions	$e ::= n \mid \texttt{true} \mid \texttt{false} \mid v \mid e_1\ aop\ e_2 \mid e_1\ bop\ e_2 \mid e_1\texttt{<=}e_2 \mid \texttt{!}e$ $\mid \texttt{fun}(td_1\ v_1, \ldots, td_n\ v_n)\{c\} \mid e_0(e_1, \ldots, e_n)$
type declarations	$td ::= \texttt{int} \mid \texttt{bool} \mid \texttt{fun}(td_1\ \texttt{*} \cdots \texttt{*}\ td_n\ \texttt{->}\ td_0)$
arithmetic operators	$aop ::= \texttt{+} \mid \texttt{-} \mid \texttt{*}$
boolean operators	$bop ::= \texttt{\&\&} \mid \texttt{||}$
variable names	v
integer constants	n
types	$\tau ::= int \mid bool \mid (\tau_1 \times \cdots \times \tau_n) \to \tau_0$
values	$u \in \mathbb{Z} \cup \{T, F\} \cup \{\lambda(v_1, \ldots, v_n).c\}$
stores	$\sigma : v \rightharpoonup u$
typing contexts	$\Gamma : v \rightharpoonup \tau \times \{T, F\}$

5.2 Large-step Semantics of TIMP

5.2.1 Commands

$$\langle \texttt{skip}, \sigma \rangle \Downarrow \sigma \tag{1}$$

$$\langle td\ v, \sigma \rangle \Downarrow \sigma \tag{2}$$

$$\frac{\langle c_1, \sigma \rangle \Downarrow \sigma_2 \qquad \langle c_2, \sigma_2 \rangle \Downarrow \sigma'}{\langle c_1; c_2, \sigma \rangle \Downarrow \sigma'} \tag{3}$$

$$\frac{\langle e, \sigma \rangle \Downarrow u}{\langle v \texttt{:=} e, \sigma \rangle \Downarrow \sigma[v \mapsto u]} \tag{4}$$

$$\frac{\langle e, \sigma \rangle \Downarrow u}{\langle v \texttt{:=} e, \sigma \rangle \Downarrow \sigma[v \mapsto u]} \tag{5}$$

$$\frac{\langle e, \sigma \rangle \Downarrow T \qquad \langle c_1, \sigma \rangle \Downarrow \sigma'}{\langle \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2, \sigma \rangle \Downarrow \sigma'} \tag{6}$$

$$\frac{\langle e, \sigma \rangle \Downarrow F \qquad \langle c_2, \sigma \rangle \Downarrow \sigma'}{\langle \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2, \sigma \rangle \Downarrow \sigma'} \tag{7}$$

$$\frac{\langle \texttt{if}\ e\ \texttt{then}\ (c; \texttt{while}\ e\ \texttt{do}\ c)\ \texttt{else skip}, \sigma \rangle \Downarrow \sigma'}{\langle \texttt{while}\ e\ \texttt{do}\ c, \sigma \rangle \Downarrow \sigma'} \tag{8}$$

5.2.2 Expressions

$$\langle n, \sigma \rangle \Downarrow n \tag{9}$$

$$\langle \texttt{true}, \sigma \rangle \Downarrow T \tag{10}$$

$$\langle \texttt{false}, \sigma \rangle \Downarrow F \tag{11}$$

$$\langle v, \sigma \rangle \Downarrow \sigma(v) \tag{12}$$

$$\frac{\langle e_1, \sigma \rangle \Downarrow n_1 \qquad \langle e_2, \sigma \rangle \Downarrow n_2}{\langle e_1 \texttt{<=} e_2, \sigma \rangle \Downarrow n_1 \leq n_2} \tag{13}$$

$$\frac{\langle e_1, \sigma \rangle \Downarrow p \qquad \langle e_2, \sigma \rangle \Downarrow q}{\langle e_1\ \texttt{\&\&}\ e_2, \sigma \rangle \Downarrow p \wedge q} \tag{14}$$

$$\frac{\langle e_1, \sigma \rangle \Downarrow p \qquad \langle e_2, \sigma \rangle \Downarrow q}{\langle e_1\ \texttt{||}\ e_2, \sigma \rangle \Downarrow p \vee q} \tag{15}$$

$$\frac{\langle e, \sigma \rangle \Downarrow p}{\langle \texttt{!}e, \sigma \rangle \Downarrow \neg p} \tag{16}$$

$$\frac{\langle e_1, \sigma \rangle \Downarrow n_1 \qquad \langle e_2, \sigma \rangle \Downarrow n_2}{\langle e_1 \texttt{+} e_2, \sigma \rangle \Downarrow e_1 + e_2} \tag{17}$$

$$\frac{\langle e_1, \sigma \rangle \Downarrow n_1 \qquad \langle e_2, \sigma \rangle \Downarrow n_2}{\langle e_1 \texttt{-} e_2, \sigma \rangle \Downarrow n_1 - n_2} \tag{18}$$

$$\frac{\langle e_1, \sigma \rangle \Downarrow n_1 \qquad \langle e_2, \sigma \rangle \Downarrow n_2}{\langle e_1 \texttt{*} e_2, \sigma \rangle \Downarrow n_1 n_2} \tag{19}$$

$$\langle \texttt{fun}(td_1\ v_1, \ldots, td_n\ v_n)\{c\}, \sigma \rangle \Downarrow \lambda(v_1, \ldots, v_n).c \tag{20}$$

$$\frac{\langle e_0, \sigma \rangle \Downarrow \lambda(v_1, \ldots, v_n).c \qquad \forall i \in 1..n\ .\ \langle e_i, \sigma \rangle \Downarrow u_i \qquad \langle c, \sigma[v_1 \mapsto u_1] \cdots [v_n \mapsto u_n] \rangle \Downarrow \sigma'}{\langle e_0(e_1, \ldots, e_n), \sigma \rangle \Downarrow \sigma'(\texttt{ret})} \tag{21}$$

5.3 Typing Rules for TIMP

5.3.1 Commands

$$\Gamma \vdash \texttt{skip} : \Gamma \tag{22}$$

$$\frac{v \notin Dom(\Gamma)}{\Gamma \vdash td\ v : \Gamma[v \mapsto (typ(td), F)]} \tag{23}$$

$$\frac{\Gamma \vdash c_1 : \Gamma_2 \qquad \Gamma_2 \vdash c_2 : \Gamma'}{\Gamma \vdash c_1; c_2 : \Gamma'} \tag{24}$$

$$\frac{\Gamma \vdash e : \tau \qquad \Gamma(v) = (\tau, p)}{\Gamma \vdash v\texttt{:=}e : \Gamma[v \mapsto (\tau, T)]} \tag{25}$$

$$\frac{\Gamma \vdash e : bool \qquad \Gamma \vdash c_1 : \Gamma_1 \qquad \Gamma \vdash c_2 : \Gamma_2}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 : \Gamma} \tag{26}$$

$$\frac{\Gamma \vdash e : bool \qquad \Gamma \vdash c : \Gamma_1}{\Gamma \vdash \texttt{while}\ e\ \texttt{do}\ c : \Gamma} \tag{27}$$

5.3.2 Non-function Expressions

$$\Gamma \vdash n : int \tag{28}$$

$$\Gamma \vdash \texttt{true} : bool \tag{29}$$

$$\Gamma \vdash \texttt{false} : bool \tag{30}$$

$$\frac{\Gamma(v) = (\tau, T)}{\Gamma \vdash v : \tau} \tag{31}$$

$$\frac{\Gamma \vdash e_1 : int \qquad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1\ aop\ e_2 : int} \tag{32}$$

$$\frac{\Gamma \vdash e_1 : bool \qquad \Gamma \vdash e_2 : bool}{\Gamma \vdash e_1\ bop\ e_2 : bool} \tag{33}$$

$$\frac{\Gamma \vdash e_1 : int \qquad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1\texttt{<=}e_2 : bool} \tag{34}$$

$$\frac{\Gamma \vdash e : bool}{\Gamma \vdash \texttt{!}e : bool} \tag{35}$$

5.3.3 Function Expressions

$$\frac{\begin{array}{c}\forall i \in 1..n\ .\ \tau_i = typ(td_i) \qquad v_1, \ldots, v_n \notin Dom(\Gamma) \\ \Gamma[v_1 \mapsto (\tau_1, T)] \cdots [v_n \mapsto (\tau_n, T)] \vdash c : \Gamma' \qquad \Gamma'(\texttt{ret}) = (\tau_0, T)\end{array}}{\Gamma \vdash \texttt{fun}(td_1\ v_1, \ldots, td_n\ v_n)\{c\} : (\tau_1 \times \cdots \times \tau_n) \to \tau_0} \tag{36}$$

$$\frac{\Gamma \vdash e_0 : (\tau_1 \times \cdots \times \tau_n) \to \tau_0 \qquad \forall i \in 1..n\ .\ \Gamma \vdash e_i : \tau_i}{\Gamma \vdash e_0(e_1, \ldots, e_n) : \tau_0} \tag{37}$$

Here, *typ* is just a function that turns a type declaration (a sequence of characters typed on your keyboard) into an equivalent type (a mathematical entity). That is, *typ* can be formally defined as:

$$\begin{aligned} typ(\texttt{int}) &= int \\ typ(\texttt{bool}) &= bool \\ typ(\texttt{fun}(td_1\texttt{*}\cdots\texttt{*}td_n\texttt{->}td_0)) &= (typ(td_1) \times \cdots \times typ(td_n)) \to typ(td_0) \end{aligned}$$

You do not have to implement *typ* because it is already implemented within the parser. (It is the part of the parser that creates OCaml values like `TypInt` that implement TIMP types.)