CS251 Programming Languages **Handout # 13**
Prof. Lyn Turbak **January 31, 2007**
Wellesley College

Problem Set 1

Due: 11:59pm Wednesday, February 7

Reading:

- Handouts #1 – #12 (only Chapters 1–5 of Handout #9 = *Introduction to the Objective Caml Programming Language*)

Overview:

In this assignment, you will (1) think about programming language implementation and (2) do some simple experimentation with OCAML. **All problems on this problem set are group problems. You are encouraged to solve them in two-person teams.**

Submission:

Each team should turn in a single hardcopy submission packet for all problems by slipping it under Lyn's office door by 11:59pm on the due date. The packet should include:

1. a team header sheet (see the end of this assignment for the header sheet) indicating the time that you (and your partner, if you are working with one) spent on the parts of the assignment.
2. your written solutions to Problems 1, 2, and 3.
3. your final version of `ps1.ml` for Problem 4.

Each team should also submit a single softcopy (consisting of your final `ps1` directory) to the drop directory `~cs251/drop/ps1/`*username*, where *username* is the username of one of the team members (indicate which drop folder you used on your hardcopy header sheet). To do this, execute the following commands in Linux in the account of the team member being used to store the code.

```
cd /students/username/cs251
cp -R ps1 ~cs251/drop/ps1/username/
```

Group Problem 0: Getting Started

Problems 1 and 2 are pencil-and-paper problems that you can do without a computer. But you will need a computer to check your answers to Problem 3 and to program your answers to Problem 4. You should complete the following steps before using the computer in Problems 3 and 4.

a. : Linux

You should begin this assignment by learning (or reminding yourself) how to use Linux. See the information in Handouts #3 and #4 on using Linux/Unix.

b. : Emacs

Next you should learn (or remind yourself) how to use Emacs. See the information in Handouts #3, #5, and #6 on using Emacs. Learning to execute all cursor-motion and editing commands via keystrokes (rather than via mouse and menus) is an important skill that will save you lots of time over the semester. It will also make it easier for you to work remotely via `ssh`. A good way to begin learning the keystroke commands is taking the online interactive Emacs tutorial (see Handout #3 for how to do this).

c. : CVS

In order to download code for your CS251 assignments, you will need to update your local copies of files that are in the CVS-controlled CS251 repository. Follow the directions in Handout #7 for how to install your local CVS filesystem. You only need to install it once.

Once you have installed your local CVS filesystem, you can access all CVS-controlled files by executing the following in a Linux shell:

```
cd ~/cs251
cvs update -d
```

Indeed, every time you log in to a Linux machine to work on a CS251 assignment, you should execute the above commands to ensure that you have the most up-to-date versions of the problem set materials.

On this assignment, executing the above commands will create the local directory `~/cs251/ps1` containing the file `ps1.ml`. This file contains simple stubs for the functions `f` and `g` you will define in Problem 4.

d. : OCAML Interpreter

For problems 3 and 4, you will need to launch the OCAML interpreter. There are several ways to do this; see Handout #10. It is recommended that you choose one of the ways that works within Emacs, since this simplifies many interactions.

Group Problem 1 [25]: Interpreters and Compilers

As seen in Lecture #1, there are two basic reasoning rules involving interpreters and translators (a.k.a. compilers):

1. *The Interpreter Rule (I)*

$$\frac{\text{P-in-L program;}\quad \text{L intepreter machine}}{\text{P machine}}\ (I)$$

2. *The Translator Rule (T)*

$$\frac{\text{P-in-S program;}\quad \text{S-to-T translator machine}}{\text{P-in-T program}}\ (T)$$

In practice, we will often elide the word "machine" for intepreter machines and translator machines. E.g., we will refer to an "L interpreter machine" as an "L interpreter", and an "S-to-T translator machine" as an "S-to-T translator". We will also often elide the word "program"; e.g., we will refer to a "P-in-L program" as "P-in-L".

a. [**10**] Suppose you are given the following:

- a Java-to-C-in-Scheme compiler (that is, a Java-to-C compiler written in Scheme);
- a Scheme-in-Pentium interpreter;
- a C-to-Pentium-in-Pentium compiler;
- a Pentium-based computer.

Using the two reasoning rules above, construct a "proof" that demonstrates how to execute a given Java program on the computer.

b. [**15**] Suppose you are given the following:

- a OCAML-in-MIPS interpreter;
- a OCAML-to-MIPS-in-OCAML compiler;
- a MIPS-based computer.

i Show how to generate a OCAML-to-MIPS-in-MIPS compiler. (Use the reasoning rules to construct a proof.)

ii After you successfully complete part i, you accidentally delete the OCAML-in-MIPS interpreter. Show how you can still execute any OCAML program on your MIPS-based computer. (Use the reasoning rules to construct a proof.)

Group Problem 2 [25]: Trusting Trust

In his paper *Reflections on Trusting Trust* (Handout #12), Ken Thompson describes how a compiler can harbor a so-called "Trojan horse" that can make compiled programs insecure. Read this paper carefully and do the following tasks to demonstrate your understanding of the paper:

a. [**10**] Section II of the paper describes how a C compiler can be "taught" to recognize `\v` as the vertical tab character. Using the same kinds of components and processes used in Problem 1, summarize the content of Section II by carefully listing the components involved and describing (by constructing a proof) how a C compiler that "understands" the code in Figure 2.2 can be generated. (Note that the labels for Figures 2.1 and 2.2 are accidentally swapped in the paper.)

b. [**15**] Section III of the paper describes how to generate a compiler binary harboring a "Trojan horse". Using the same kinds of components and processes used in Problem 1, summarize the content of Section III by carefully listing the components involved and describing (by constructing a proof) how the Trojaned compiler can be generated.

Group Problem 3 [20]: OCAML Evaluation

Fig. 1 shows a sequence of OCAML declarations and expressions. Show the type and value of each declaration and expression, assuming that they are evaluated sequentially. You should predict the answers without using the OCAML interpreter, but may use it to check your answers.

Group Problem 4 [30]: OCAML Functions

Consider the following Java class method:

```
public static int f (int n) {
  if (n <= 2) {
    return n;
  } else {
    return n + f(n/2) + f(n/3);
  }
}
```

Fig. 2 shows an invocation tree for the invocation `f(30)`. In the tree, each node labeled `f(`*n*`):`*r* denotes an invocation of the method `f` on the argument *n* that yields the result *r*.

a. [**10**]: **Translating `f` to OCAML**

In the file `ps1.ml` (see Problem 0(c) for how to create this), flesh out an OCAML declaration for the function `f`.

Within the OCAML interpreter, execute the following to load `ps1.ml` the first time:

```
let a = 5;;

let b = a * a;;

2*a + (b-a);;

let avg = fun x y -> (x+y)/2;;

let avg a b = (a+b)/2;;

avg (2*a) (b-a);;

let app_3_5 = fun f -> f 3 5;;

app_3_5 (+);;

app_3_5 (-);;

app_3_5 ( * );;

app_3_5 avg;;

app_3_5 (fun x y -> x);;

app_3_5 (if 1>2 then (+) else ( * ));;

app_3_5 (fun x y -> if x < y then (+) else (-));;

let add_a x = x + a;;

add_a 100;;

let a = 17;;

b;;

add_a 100;;

let try_a a = add_a (2*a);;

try_a 100;;
```

Figure 1: A sequence of OCAML declarations and expressions.

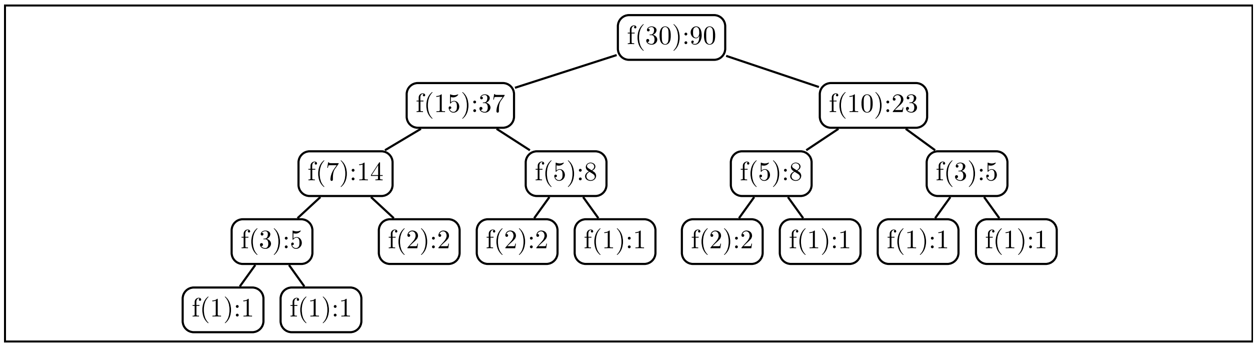

Figure 2: Invocation tree for `f(30)`.

```
# #cd "/students/username/cs251/ps1";;
# #use "ps1.ml";;
```

Note that the first hash mark (#) in each line is the prompt of the OCAML interpreter, while the second hash mark in each line is part of the command name. For some reason, OCAML does not understand the abbreviation ~ for `/students/`*username*, so you must write out the long form. When `ps1.ml` loads properly, you should see the following:

```
val f : int -> int = <fun>
val g : int -> int * int = <fun>
```

You are now ready to test you definition of `f`. The following is a transcript of some tests for a working `f`:

```
# f 1;;
- : int = 1
# f 2;;
- : int = 2
# f 3;;
- : int = 5
# f 5;;
- : int = 8
# f 7;;
- : int = 14
# f 10;;
- : int = 23
# f 15;;
- : int = 37
# f 30;;
- : int = 90
```

If your definition of `f` contains an error, you will need to edit `ps1.ml`. After making any changes to `ps1.ml`, you need to re-execute

```
# #use "ps1.ml";;
```

to inform the OCAML interpreter of your changes.

b. [20]: Counting nodes

In this part your goal is to define an OCAML function `g` that takes an integer argument *n* and returns a *pair* of numbers `(`*r*`,`*c*`)`, where *r* is the result of `f(`*n*`)` and *c* is the count of the number of `f` nodes in the invocation tree for `f(`*n*`)`. For example, `f(30)` yields `90` and the number of nodes in the invocation tree for `f(30)` is `17`, so `g(30)` should return the pair `(90,17)`.

To achieve full credit for this part, you must define `g` as a single recursive function that does *not* call the function `f` and does not use any local functions. However, partial credit will be awarded for definitions of `g` that invoke `f`.

The following is a transcript of some tests for a working `g`:

```
# g 1;;
- : int * int = (1, 1)
# g 2;;
- : int * int = (2, 1)
# g 3;;
- : int * int = (5, 3)
# g 5;;
```

```
- : int * int = (8, 3)
# g 7;;
- : int * int = (14, 5)
# g 10;;
- : int * int = (23, 7)
# g 15;;
- : int * int = (37, 9)
# g 30;;
- : int * int = (90, 17)
```

Extra Credit 1 [25]: Self-printing Program

In his article *Reflections on Trusting Trust*, Ken Thompson notes that it is an interesting puzzle to write a program in a language whose effect is to print out a copy of the program. We will call this a *self-printing program*. Write a self-printing program in any general-purpose programming language you choose. Your program should be self-contained – it should not read anything from the file system.

Problem Set Header Page
Please make this the first page of your hardcopy submission.

CS251 Problem Set 1
Due 11:59pm Wednesday, February 7

Names of Team Members:

Date & Time Submitted:

Collaborators (*anyone you or your team collaborated with on the problem set*):

In the **Time** *column, please estimate the time you or your team spent on the parts of this problem set. Team members should be working closely together, so it will be assumed that the time reported is the time for each team member. Please try to be as accurate as possible; this information will help me design future problem sets. I will fill out the* **Score** *column when grading your problem set.*

Part	**Time**	**Score**
General Reading		
Problem 1 [25]		
Problem 2 [25]		
Problem 3 [20]		
Problem 4 [30]		
Total		