CS301 Compiler Design and Implementation **Handout # 11**
Prof. Lyn Turbak **September 17, 2003**
Wellesley College

Problem Set 2
Due: Friday, September 26

Reading:

- Handout #9: INTEX: An Introduction to Program Manipulation
- Handout #10: POSTFIX: A Simple Stack Language
- Handout #12: Testing
- Handout #13: S-Expressions, Parsing, and Unparsing

Submission:
Each team should turn in a single hardcopy submission packet for all problems by slipping it under Lyn's office door by 5pm on the due date (Fri, Sep 26). The packet should include:

1. a team header sheet (see the end of this assignment for the header sheet);
2. your final version of `PostFixInterp.ml` from Problem 1.
3. your final version of `PostFix.ml` from Problem 2.
4. your final version of `PostFixInterpTest.ml` from Problem 3.

Each team should also submit a softcopy submission of the final `ps2` folder. To do this, execute the following commands in Linux in the account of the team member being used to store the code.

```
cd /students/your-account-name/cs301
cp -R ps2 /home/cs301/drop/your-account-name/
```

High-Level Notes:

- If you work on a team for this assignment, you must choose a different partner than for PS1.
- Excluding Jue, there are an odd number of students in the class, which does not permit everyone to work in teams of two. Kelsey, who worked alone on PS1, has first dibs on a partner for PS2. I will consider a three-person team if it is proposed, but keep in mind that scheduling meetings for 3-person teams may be very difficult!
- Although the problem set is phrased in terms of three separate problems, they are all part of the same big problem (implementing a POSTFIX interpreter). In particular, you do *not* need to complete them in order. Indeed, you will probably want to interleave work on the problems. E.g., implement an interpreter for a simple subset of the language, test this interpreter, and maybe implement a parser/unparser for this subset before handling more features.
- When editing OCAML code in Emacs, it is helpful to be in "CAML mode". You can enter CAML editing mode in a editor buffer via `M-x caml-mode`.

Problem 1 [60]: A PostFix Interpreter

Background:

In this problem, you will implement an interpreter for the PostFix language described in Handout #10. As explained there, the abstract syntax of PostFix can be expressed in OCaml via the following two datatypes for programs (`pgm`) and commands (`com`), which are already defined for you in the module `PostFix` in the file `PostFix.ml`.[1]

```
type pgm = (* PostFix programs *)
    Pgm of int * com list

and com = (* PostFix commands *)
    Int of int     (* push integer literal *)
  | Str of string  (* push string literal *)
  | Seq of com list (* executable sequence *)
  | Pop  (* pop top value from stack *)
  | Swap (* swap top two values of stack *)
  | Sel  (* choose one of two values from stack *)
  | Get  (* push value at given stack index *)
  | Put  (* store top of stack at given stack index *)
  | Prs  (* print string *)
  | Pri  (* print integer *)
  | Exec (* execute sequence at top of stack *)
  | Add | Sub | Mul | Div | Rem (* arithmetic ops *)
  | LT | LE | EQ | NE | GE | GT (* relational ops *)
```

You will implement your interpreter in the module `PostFixInterp`. In addition to opening up the `PostFix` module to make the syntax datatypes available, the `PostFixInterp` module defines the following dataypes needed for the interpreter:

1. The interpreter manipulates a stack of values that may either be integers, strings, or executable sequences. These stack values are modeled by the `sval` datatype:

   ```
   type sval =  (* stack values *)
       IntVal of int
     | StrVal of string
     | SeqVal of com list
   ```

2. The "abstract machine state" manipulated by the PostFix interpreter consists of two parts: (1) a list of commands and (2) a stack of values. In OCaml, an easy way to model the stack is as a list. So the following `state` type represents the configuration manipulated by the interpreter:

   ```
   type state = com list * sval list
   ```

   Note that this is not like other type definitions we have seen – it does not specify a constructor! A constructorless type definition introduces a name that abbreviates a type. So `state` is just a shorter name for the pair type `com list * sval list`.

3. The result of executing a PostFix program is either an integer or an error. This is modeled by the `ans` datatype:

[1] We will follow the convention that modules are stored in files with the module name followed by a `.ml` suffix.

```
type ans = (* answers to executing PostFix programs *)
    IntAns of int    (* integer answer *)
  | ErrAns of string (* error answer *)
```

In the PostFix interpreter, it is possible to model all error situations using the ErrAns constructor. In contrast with the Intex interpreter, it is *not* necessary to use exceptions to model any error situations.

Your Task:

To complete the PostFix interpreter, you will need to define two functions:

- `val exec : state -> ans`
  Given a state `(coms,svals)`, the `exec` function executes all of the PostFix commands `coms` in the context of a stack consisting of the values `svals`. It returns an answer that is either the integer at the top of the final stack or an error describing a state that is "stuck" (i.e., a non-final state from which no progress can be made).

- `val run : PostFix.pgm -> int list -> ans`
  Executes the given PostFix program on an integer list of arguments.

Your error messages should be chosen to accurately describe the type of error encountered. There are *lots* of different error situations; make sure you handle them all!

Notes:

- As usual, start this problem set by performing a `cvs update -d`, and perform an update every time you log in to work on this problem set.

- After launching OCaml, connect to the `ps2` directory via `#cd "/students/`*your-account-name*`/ps2"`.

- To load all the appropriate files into OCaml, execute `#use "load-postfix.ml"`. You will need to execute this every time you want to test changes to your code. This loads many files, including `PostFix.ml`, `PostFixInterp.ml`, and `PostFixInterpTest.ml`. Carefully look at the output of the `#use` command each time you invoke it. Even if it finds and reports an error in `PostFix.ml`, say, it will continue to load the other files, and it's easy to miss the error, especially if the error message has scrolled off the screen.

- The `#trace` directive can be *very* helpful for debugging a non-working `exec` function. The tracing output is easiest to read if you invoke it as follows:

  ```
  # open PostFix;;
  # open PostFixInterp;;
  # #trace exec;;
  ```

- You *should* use any standard library functions from the standard modules that you find helpful.

- You should *not* use any imperative features of OCaml (e.g. reference cells, array) to implement your interpreter. Even though the `state` type is immutable (consisting of two immutable lists), you can simulate changing state by creating a sequence of new state pairs in a tail recursion rather than modifying existing ones in a loop.

- Pattern matching is your friend here. Use it to simplify your definitions.
- In addition to defining `exec` and `run`, you may need to define some auxiliary functions.
- When implementing the commands `Prs` and `Pri`, use `StringUtils.unbufferedPrint` instead of `print_string`. The problem with `print_string` is that it "buffers" strings (potentially for a long time) before printing them. If you want to ensure that something gets printed on the console immediately, use `StringUtils.unbufferedPrint`.
- When printing values, it is necessary to sequence expressions. There are two styles of evaluating expressions $e_1, e_2, \ldots, e_n$ in order in OCaml:

  1. In the *semi-colon style*, semi-colons are used to sequence statements (like in Java and C). It is often necessary to parenthesize the sequenced expressions as well. E.g.

     `(`e_1`;`e_2`;`$\ldots$`;`e_n`)`

     The value of such a sequence is the value of the last expression (e_n).
  2. In the *let style*, `let`s are used to sequence the expressions used to sequence statements (like in Java and C). It is often necessary to parenthesize the sequenced expressions as well. E.g.

     ```
     let _ = e1 in
     let _ = e2 in
       ⋮
     let _ = en-1 in
       en
     ```

- It's a good idea to implement, test, and debug a few commands at a time rather than attempting to handle all the commands at once.
- You can test `exec` and `run` before you have completed Problem 2 by writing programs in terms of abstract syntax rather than concrete syntax.

Problem 2 [15]: Parsing and Unparsing PostFix

Although it is possible to write all PostFix programs and commands using the OCaml AST constructors, it is inconvenient to do so. In this part, you will implement the concrete s-expression syntax for PostFix described in Handout #10 by implementing the following four functions in the `PostFix` module:

- `val sexp2Com : Sexp.sexp -> com`
  Parse an s-expression representation of a PostFix command into a value of type `com`.
- `val sexp2Pgm : Sexp.sexp -> pgm`
  Parse an s-expression representation of a PostFix program into a value of type `pgm`.
- `val pgm2Sexp : pgm -> Sexp.sexp`
  Unparse a value of type `pgm` into an s-expression representation of a PostFix program.
- `val com2Sexp : com -> Sexp.sexp`
  Unparse a value of type `com` into an s-expression representation of a PostFix command.

Notes:

- Study the Intex examples of parsing and unparsing s-expressions in Handout #13 (and `Intex.ml`) before attempting this problem.
- Because some `sexp` and `com` constructors have the same names (e.g. `Int`, `Str`, `Seq`), you will need to qualify at least one set of constructor names with an explicit module name. It's simplest to use an explicit `Sexp.` prefix for all s-expression constructors.
- When a parser encounters an error, it should raise a `SyntaxError` exception with a single string argument.
- Once the four above functions are defined, the following four functions that convert between strings and PostFix programs and commands are easy to define:

```
let string2Com str = sexp2Com (Sexp.string2Sexp str)
let string2Pgm str = sexp2Pgm (Sexp.string2Sexp str)
let com2String c = Sexp.sexp2String (com2Sexp c)
let pgm2String p = Sexp.sexp2String (pgm2Sexp p)
```

Problem 3 [25]: Testing your PostFix interpreter

In this part, you will create an extensive test suite for your PostFix interpreter to show that it behaves correctly in a wide variety of circumstances. In the module `PostFixInterpTest` you should write a function `val test : unit -> unit` that exercises your PostFix interpreter on the test suite.

Your test suite should include the following:

- All of the examples from Handout #10 (except the infinite loop example, of course).
- A two-argument PostFix program that implements the following OCaml `gcd` function:

  ```
  let rec gcd a b =
    if b = 0 then
      a
    else
      gcd b (a mod b)
  ```

  As part of this problem, you will need to figure out how to express a recursive function in PostFix. (Hint: study the factorial example in Handout #10 and think *very* carefully about the problem.) You should test your `gcd` program on a wide variety of inputs.
- Programs that cause each one of the error situations handled by your interpreter. That is, you should make sure that your test suite "exercises" each error condition caught by your interpreter.

Notes:

- Study Handout #12 and the Intex testing suite in `intex/IntexInterpTest.ml` before doing extensive testing.
- Although you can use abstract syntax for simple testing, it would be too onerous to use it for all testing. It is better to complete Problem 2 before doing extensive testing.
- When using string literals to write down PostFix programs that themselves contain strings, be careful to use escape characters for embedded double quotes. For instance, the PostFix program

  ```
  (postfix 0 "foo" prs)
  ```

  is written as a string literal as follows:

  ```
  "(postfix 0 \"foo\" prs)"
  ```

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

CS301 Problem Set 2

Due Friday, September 26

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 you problem set.*

Part	**Time**	**Score**
General Reading		
Problem 1 [60]		
Problem 2 [15]		
Problem 3 [25]		
Total		