CPSC 201: Introduction to Computer Science
Carsten Schürmann
Date: March 27, 2002

Homework 6

Due: midnight
Wednesday April 10, 2002.

In this assignment, we will implement a lexer, a parser, and an interpreter for a small programming language, called "Fun". This sounds like a big project, and it kind of is, so *do not wait until the last minute with starting the homework, start it early.* Fun resembles a little bit SML, but we don't allow functions to be passed as values. Fun's features include user defined function, integers, floating point numbers, and Booleans. This is an implementation of the factorial function in Fun.

```
fun fac x = if x = 0 then 1 else x * (fac (x-1))
```

Consider Fun's syntax of programs which is given in EBNF form.

$$
\begin{array}{lcl}
e & ::= & i\ e' \mid n\ e' \mid x\ e' \mid \texttt{true} \mid \texttt{false} \\
 & & \mid \texttt{fun}\ i_1\ i_2 = e \mid \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \mid \texttt{not}\ e \mid (e) \\
e' & ::= & \epsilon \mid +e_2 \mid -e_2 \mid *e_2 \mid /e_2 \mid =e_2 \mid <e_2 \mid >e_2 \mid (e)
\end{array}
$$

In this definition the terminal symbols are i (for identifiers, such as variable names, and function names), n (for integers), x (for floating point numbers). ϵ stands for "nothing". The two non-terminal symbols are e and e'. For your convenience, we have given the grammar rules already in the right form (this is why they look a little funny), so it is straightforward to implement the recursive descent parser.

The relevant code provided by us for this assignment can be found in the course directory under `/c/cs201/lib/code/ass6` on the Zoo.

Problem 1: Lexer (20 points)

Based on the code we showed in class, write a lexer for Fun.

```
lexer: string -> Token list
```

To this end, you will have to define your own tokens. We suggest you to extend the ones presented in class. For example: the result of lexing

```
fun succ x = x+1
```

should result in the following list of tokens.

```
[FUN,ID "succ",ID "x",EQ,ID "x",PLUS,INTEGER 1] : Token list
```

Problem 2: Internal Representation (20 points)

Develop an internal representation for Fun programs.

```
datatype Exp = ...
```

that reflects the grammar rules from above. Hint: You could write two mutually recursive datatype definitions `Exp` and `Exp'` for e and e', respectively, but this is *not recommended*. Just define one single one, called `Exp`. The difference, between the two solutions is that with only `Exp`, you could theoretically represent many more Fun programs than are parsable with the grammar above, but this is quite alright. Example: the Fun program

```
fun succ x = x+1
```

should be parsed into the following ML object.

```
Fun ("succ","x",Plus (Var "x",Int 1)) : Exp
```

Problem 3: Printing (20 points)

Write a function

```
expToString : Exp -> string
```

that converts a Fun program into a string that can be printed for inspection.

Problem 4: Parser (20 points)

Using your solution from Problem 1, write a parser for Fun. Your parser should be implemented based on the recursive descent technique. As input, it takes in a list of tokens, and returns a Fun program. You probably want to write two (mutual recursive) functions. One that parses expressions e, and another that parses expressions e'.

```
parse: Token list -> Exp * Token list
parse': Exp * Token list -> Exp * Token list
```

`parse ts = (E, ts')` satisfies the following invariant: `ts` is a list of tokens to be parsed, `E` is a Fun program returned that corresponds to the initial segment to `ts`, and `ts'` is the list of tokens that have not been used yet. `parse' (E, ts) = (E', ts')` on the other hand, has one additional input argument, `E`, which is the identifier i, n, or x, of the "head" of the expression currently parsed. See the three first production rules of e. Note, that for simplicity we ignore operator precedence. For example: parsing the list of tokens from the example above

```
parser [FUN,ID "succ",ID "x",EQ,ID "x",PLUS,INTEGER 1]
```

should result in the following:

```
(Fun ("succ","x",Plus (Var "x",Int 1)),[]) : Exp * Token list
```

The left expression of this pair is the object to be parsed, the right is a list of remaining tokens, which is empty in this particular example.

Problem 5: Evaluator (20 points)

Using your solution from Problem 1, write an evaluator for Fun. When it evaluates, programs, it has to store bindings some place in an environment which is simply a list of pairs. A pair consists of the variable name and the value it is bound to.

```
type env = (string * Exp) list
```

The following functions allow you to lookup and update the environment. `empty` is the empty environment.

```
val empty = []
fun lookup s ((s', v) :: env) = if s = s' then v else lookup s env
fun update (s, v) env = (s, v) :: env
```

The evaluator which you are about to write takes a Fun program of type `Exp` as input, and returns a value (also a Fun expression) of type `Exp`.

```
eval : env -> Exp -> Exp
```

To test your program, use the top level loop that we have provided, and run it. You can start it with `top ()` from the ML top level. The following is an example session with the Fun interpreter.

```
Fun (Version 1.0, Yale University)
> fun succ x = x+1
> succ (41)
Answer: 42
> fun fac x = if x = 0 then 1 else x * (fac (x-1))
> fac (10)
Answer: 3628800
```

Hand-in Instructions

In the course directory /c/cs201/bin, there are five programs that support you in submitting your solution to the homework.

```
submit    assignment-number file(s)
unsubmit  assignment-number file(s)
check     assignment-number
protect   assignment-number file(s)
unprotect assignment-number file(s)
```

`submit` allows you to submit one of several files. For example

```
/c/cs201/bin/submit 6 hw6.sml hw6-examples.sml
```

submits your file `hw6.sml` and `hw6-examples.sml`. `check` can give you the peace of mind that your solution has really been submitted, and if you would like to make last minute changes to your solution, use `unsubmit` before submitting the updated version. `protect` and `unprotect` give you the power to protect/unprotect submitted solutions from deletion.