

Design and Selection of Programming Languages

28 October 2005

Exercise 7.1

Use the interaction framework from Exercise Sheet 4 and extend the parsers from the lecture to be able to produce `ghci`-like interaction:

```
let k = 3 * 4
> k = 12
let n = 3 * (k + 2)
> n = 42
n - k
> 30
n + m
> error: Variable "m" undefined!
```

Hints:

- Determine the input grammar for this system.
- Define a datatype to represent the ASTs of these inputs.
- Define a parsing function for these inputs.
- The current version of *SimpleLexer* on the course page can be used, but is of course not ideal. Starting from a variant of the lexer from Exercise 6.4 would be better.
- Determine an appropriate state datatype for this interaction.
- You will need an evaluation function for expressions with variables; this will of course need a variable assignment.
- Define the *Transition* function for this interaction.
- Define a *main* computation (using *runprocess* from Sheet 4) and compile your program.
Test both the interpreted and the compiled versions.

Solution Hints

For simplicity, we use here *SimpleLexer* and no advanced Haskell features.

In fact, many aspects could be done nicer, or “more naturally”, and more generally. But the goal of the presented solution is that it should be generally understandable, and at a level you could produce yourselves.

For convenience, we define a type synonym for variables:

```
type Variable = String
```

The abstract syntax representation of inputs is a datatype with two variants, one for let bindings, and one for expressions that should be evaluated:

```

data Command
  = Let Variable Expr
  | Eval Expr

```

Using the existing parser for expressions, we easily get a parser for let bindings:

```

parseLet :: [Token] → Maybe (Command, [Token])
parseLet (Ident "let" : Ident v : Sep '=' : rest) =
  case parseExpr rest of
    Nothing → Nothing
    Just (e, rest') → Just (Let v e, rest')
parseLet _ = Nothing

```

When parsing commands, we have to try let bindings first, since our token type has no keyword variant, and the lexer therefore treats "let" just as an identifier.

```

parseCommand :: [Token] → Maybe (Command, [Token])
parseCommand toks = case parseLet toks of
  Nothing → fmap (pupd1 Eval) $ parseExpr toks
  r → r

```

Evaluating expressions containing variables needs a variable assignment, i.e., a partial function from variables to values. We introduce a type synonym for this, deciding to use *Map.Map* for the implementation of these partial functions:

```

type VarAssign = Map.Map Variable Integer

```

Since we do not want our interpreter to crash, we have to catch errors in expression evaluation and explicitly return *Left message* (for some appropriate *message :: String*) to pass the error on to the caller; successful evaluation returns *Right value* (with the appropriate *value :: Integer*). The only kind of error produced by expression evaluation itself is the use of undefined variable; otherwise, expression evaluation is then straight-forward.

```

evalExpr :: VarAssign → Expr → Either String Integer
evalExpr va (Num n) = Right n
evalExpr va (Var v) = case Map.lookup v va of
  Nothing → Left $ "Variable \"" ++ v ++ "\" undefined!"
  Just k → Right k
evalExpr va (Bin e1 op e2) = case evalExpr va e1 of
  Left e → Left e
  Right m → case evalExpr va e2 of
    Left e → Left e
    Right n → evalOp op m n

```

Among the supported operators, only division is not total, and it is easy to check the arguments for legality, since it fails exactly iff the second argument is zero:

```

evalOp :: Op → Integer → Integer → Either String Integer
evalOp (MkOp "+") m n = Right (m + n)
evalOp (MkOp "-") m n = Right (m - n)
evalOp (MkOp "*") m n = Right (m * n)
evalOp (MkOp "/") m 0 = Left "division by zero!"
evalOp (MkOp "/") m n = Right (m `div` n)

```

```
evalOp (MkOp s) _ _ = Left $ "unknown operator "" + s + """
```

For the integration of expression evaluation into the interaction loop, we have to decide on an appropriate state space — variable assignments are sufficient:

```
type State = VarAssign
```

The state transition function takes a state and an input as arguments, and produces a new state together with an output as its result.

All errors will be reflected in the output, and, in this simple case, not induce a state change. Expression evaluation does not induce a state change, either — our expressions have no side effects. Let bindings do produce a state change, namely adding the new binding to the state.

```
trInterp1 :: Transition State String String
```

```
trInterp1 (s, input) = case parseCommand (simpleLexer input) of
```

```
  Nothing → (s, "<<< syntax error >>>")
```

```
  Just (c, toks≅(_ : _)) →
```

```
    (s, "<<< spurious trailing material: " + show toks)
```

```
  Just (Let v e, []) → case evalExpr s e of
```

```
    Left e → (s, "> error: " + e)
```

```
    Right n → (Map.insert v n s, "> " + v + " = " + show n)
```

```
  Just (Eval e, []) → (s, case evalExpr s e of
```

```
    Left e → "> error: " + e
```

```
    Right n → "> " + show n
```

```
)
```

Using the machinery from Sheet 4, this directly induces an interactive program:

```
main :: IO ()
```

```
main = runprocess trInterp1 Map.empty
```

Compile this file with “ghc --make -o Interpreter1 Interpreter1.hs” (make sure that you have the latest versions of SimpleLexer.hs, Expr.hs, and StackMachine.hs) and have fun!
