Mini-Project 1: Your First Interpreter

CAS CS 320: Principles of Programming Languages

Due April 3, 2025 by 8:00PM

In this project, you'll be building an interpreter for a small *untyped* subset of OCaml. This will mean building a parser and an evaluator. There will be no type checker for this project. Your task is to implement the following functions.

```
val parse : string -> prog option
val subst : value -> string -> expr -> expr
val eval : expr -> (value, error) result
val interp : string -> (value, error) result
```

Your implementations should appear in a file `interp1/lib/interp1.ml`. The types used in the above signature appear in the module `Utils` and are detailed below. **Please read the following instructions completely and carefully.**

Part 1: Parsing

A program in our language is given by the following grammar.

```
<prog> ::= <expr>
<expr> ::= if <expr> then <expr> else <expr>
         | let <var> = <expr> in <expr>
         | fun <var> -> <expr>
         | <expr2>
<expr2> ::= <expr2> <bop> <expr2>
          | <expr3> {<expr3>}
<expr3> ::= () | true | false
          | <num> | <var>
          | ( <expr> )
 <bop> ::= + | - | * | / | mod | < | <= | > | >= | = | <> | && | ||
 <num> ::= handled by lexer
 <var> ::= handler by lexer
```

We'll use the following regular expressions in our lexer to represent whitespace, numbers and variables.

```
let whitespace = [' ' '\t' '\n' '\r']+
let num = '-'? ['0'-'9']+
let var = ['a'-'z' '_'] ['a'-'z' 'A'-'Z' '0'-'9' '_' '\'']*
```

The first regular expression says that whitespace is a nonempty sequence of spaces, tabs, or newlines. Like OCaml, our language is whitespace agnostic, so the lexer should eliminate whitespace between tokens. The next regular expression says that a number is a least one digit from `0` to `9` preceded optionally by a negation sign. Note that this means `-000` is a valid integer (like in OCaml). The last regular expression says that a variable is an alphanumeric string with underscores and apostrophes, which must start with a lowercase letter or an underscore. Note that this means that `___'''___` is a valid variable name (like in OCaml). You can used these identifiers without modification in your lexer.

Finally, in the following table we present the operators and their associativity *in order of increasing precedence*.

Operator	Associativity
`||`	right
`&&`	right
`<`, `<=`, `>`, `>=`, `=`, `<>`	left
`+`, `-`	left
`*`, `/`, `mod`	left

Your implementation of `parse` should return `None` in the case that the input string is not recognized by this grammar. It should target the following ADT, which appears in the module `Utils`.

```
type bop =
  | Add | Sub | Mul | Div | Mod
  | Lt | Lte | Gt | Gte | Eq | Neq | And | Or

type expr =
  | Num of int
  | Var of string
  | Unit
  | True | False
  | App of expr * expr
  | Bop of bop * expr * expr
  | If of expr * expr * expr
  | Let of string * expr * expr
  | Fun of string * expr

type prog = expr
```

Part 2: Evaluation

The evaluation of a program in our language is given by the big-step operational semantics below. As usual, $e \Downarrow v$ denotes that the expression e evaluates to the value v. We take a value to be one of the following.

- ▷ an integer (an element of the set $\mathbb{Z}$) denoted 1, -234, 12, etc.
- ▷ a Boolean value (an element of the set $\mathbb{B}$) denoted $\top$ and $\bot$
- ▷ unit, denoted $\bullet$
- ▷ a function, denoted $\lambda x.e$ where x is a variable and e is an expression

Values are represented by the following ADT.

```
type value =
  | VNum of int
  | VBool of bool
  | VUnit
  | VFun of string * expr
```

The value of `eval` e should be `Ok` v if $e \Downarrow v$ is derivable according to the given semantics. Otherwise it should return `Error` err where err is one of the following forms:

- ▷ `DivByZero`, the second argument of a division operator was 0
- ▷ `InvalidIfCond`, the condition in an if-expression does not evaluate to a Boolean value
- ▷ `InvalidArgs`, an operands of an operator evaluate to values of the wrong type
- ▷ `InvalidApp`, a non-function value was used in a function application expression

It will be up to you to determine which error to return when. Note that it is also possible that `eval` e does not terminate.

The remainder of this section is a formal description of the big-step operational semantics of our language. With regards to order of evaluation, **we evaluate expressions in premises from left to right**. This is important because the order of evaluation may affect what error is observed or even whether or not evaluation terminates.

Literals

$$\frac{\texttt{n} \text{ is an integer literal}}{\texttt{n} \Downarrow n}\ (\text{intEval}) \qquad \frac{}{\texttt{true} \Downarrow \top}\ (\text{trueEval}) \qquad \frac{}{\texttt{false} \Downarrow \bot}\ (\text{falseEval}) \qquad \frac{}{\texttt{()} \Downarrow \bullet}\ (\text{unitEval})$$

Arithmetic Operators

All arithmetic operators on values have their usual definitions (e.g., '+' is integer addition). Also, all of these rules have implicit side conditions of the form $v_1 \in \mathbb{Z}$ and $v_2 \in \mathbb{Z}$ since these must hold for a side condition using an arithmetic operator (e.g., $v_1 + v_2 = v$) to hold.

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad v_1 + v_2 = v}{e_1 \texttt{ + } e_2 \Downarrow v}\ (\text{addEval}) \qquad \frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad v_1 - v_2 = v}{e_1 \texttt{ - } e_2 \Downarrow v}\ (\text{subEval})$$

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad v_1 \times v_2 = v}{e_1 \texttt{ * } e_2 \Downarrow v}\ (\text{mulEval}) \qquad \frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad v_2 \neq 0 \qquad v_1 / v_2 = v}{e_1 \texttt{ / } e_2 \Downarrow v}\ (\text{divEval})$$

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad v_2 \neq 0 \qquad v_1 \ mod\ v_2 = v}{e_1 \texttt{ mod } e_2 \Downarrow v}\ (\text{modEval})$$

Comparison Operators

All comparison operators on values have their usual definitions. We assume that **comparisons can only be made between two integers**. So, as with arithmetic operators, there are implicit side conditions of the form $v_1 \in \mathbb{Z}$ and $v_2 \in \mathbb{Z}$. We present only a subset of all semantic rules for comparison. You should be able to extrapolate how the rules look for the remaining comparison operators.

$$\frac{e_1 \Downarrow v_1 \qquad e_1 \Downarrow v_2 \qquad v_1 < v_2}{e_1 \texttt{ < } e_2 \Downarrow \top}\ (\text{ltEvalTrue}) \qquad \frac{e_1 \Downarrow v_1 \qquad e_1 \Downarrow v_2 \qquad v_1 \geq v_2}{e_1 \texttt{ < } e_2 \Downarrow \bot}\ (\text{ltEvalFalse})$$

$$\frac{e_1 \Downarrow v_1 \qquad e_1 \Downarrow v_2 \qquad v_1 \leq v_2}{e_1 \texttt{ <= } e_2 \Downarrow \top}\ (\text{lteEvalTrue}) \qquad \frac{e_1 \Downarrow v_1 \qquad e_1 \Downarrow v_2 \qquad v_1 > v_2}{e_1 \texttt{ <= } e_2 \Downarrow \bot}\ (\text{lteEvalFalse})$$

$$\frac{e_1 \Downarrow v_1 \qquad e_1 \Downarrow v_2 \qquad v_1 > v_2}{e_1 \texttt{ > } e_2 \Downarrow \top}\ (\text{gtEvalTrue}) \qquad \frac{e_1 \Downarrow v_1 \qquad e_1 \Downarrow v_2 \qquad v_1 \leq v_2}{e_1 \texttt{ > } e_2 \Downarrow \bot}\ (\text{gtEvalFalse})$$

Boolean Operators

These operators behave differently than the ones above. These rules describe the standard *short-circuiting* behavior of Boolean operators.

$$\frac{e_1 \Downarrow \bot}{e_1 \texttt{ \&\& } e_2 \Downarrow \bot}\ \text{(andEvalFalse)} \qquad \frac{e_1 \Downarrow \top \qquad e_2 \Downarrow v \qquad v \in \mathbb{B}}{e_1 \texttt{ \&\& } e_2 \Downarrow v}\ \text{(andEvalTrue)}$$

$$\frac{e_1 \Downarrow \top}{e_1 \texttt{ || } e_2 \Downarrow \top}\ \text{(orEvalTrue)} \qquad \frac{e_1 \Downarrow \bot \qquad e_2 \Downarrow v \qquad v \in \mathbb{B}}{e_1 \texttt{ || } e_2 \Downarrow v}\ \text{(orEvalFalse)}$$

Conditionals

$$\frac{e_1 \Downarrow \top \qquad e_2 \Downarrow v}{\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \Downarrow v}\ \text{(ifTrueEval)} \qquad \frac{e_1 \Downarrow \bot \qquad e_3 \Downarrow v}{\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \Downarrow v}\ \text{(ifFalseEval)}$$

Variables and Functions

The implementations of these rules will depend on `subst`. We will only evaluate expressions which are *well-scoped* (i.e., have no free variables), so you **don't have to worry about implementing capture-avoiding substitution**. See the next section for more details.

$$\frac{e_1 \Downarrow v_1 \qquad [v_1/x]e_2 = e \qquad e \Downarrow v}{\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 \Downarrow v}\ \text{(letEval)}$$

$$\frac{}{\texttt{fun } x \texttt{ -> } e \Downarrow \lambda x.e}\ \text{(funEval)} \qquad \frac{e_1 \Downarrow \lambda x.e \qquad e_2 \Downarrow v_2 \qquad [v_2/x]e = e' \qquad e' \Downarrow v}{e_1\ e_2 \Downarrow v}\ \text{(appEval)}$$

Putting Everything

After you're done with `parse` and `eval`, you should combines these into a single function called `interp`. This should, in essence be function composition, with two additional contraints:

- ▷ If `parse` e fails, then `interp` e should be `Error ParseFail`
- ▷ If e has a free variable, then `interp` e should be `Error (UnknownVar` v`)` where v is the *first* free variable appearing in the input program

Once you've completed `interp`, you should be able to run

```
dune exec interp1 filename
```

in order to execute a program written in `filename` (replace `filename` with the name of the file which the program you want to execute). There is a small number of examples in the directory `examples`, but you should write more programs yourself to test your implementation. Our language is subset of OCaml so you should be able to easily write programs, e.g., here is an implementation of `sum_of_squares`:

```
let sum_of_squares = fun x -> fun y ->
  let x_squared = x * x in
  let y_squared = y * y in
  x_squared + y_squared
in sum_of_squares 3 (-5)
```

Our executable prints out the value of the expression (in this case, `34`).

Extra Credit: Recursion

The semantics we've presented here makes it difficult to implement recursion. Like OCaml, we presume that variable definitions *shadow* existing definitions, so if you try to evaluate the following program, you'll get an error saying that `fact` is an unknown variable.

```
let fact = fun n ->
  if n <= 0
  then 1
  else n * fact (n - 1)
in fact 5
```

The extra credit task is as follows:

- ▷ Update your lexer and parser to include an *optional* keyword `rec` which can be used to mark recursive functions, e.g.,

  ```
  let rec fact = fun n ->
    if n <= 0
    ...
  ```

- ▷ Read the Wikipedia page on fix-point combinators, in particular the section on recursive definitions. Use the ideas there to implement recursive functions.

It is important that, if you do the extra credit, it *does not* affect the interpreter's behavior in any other way. You should be able to implement the extra credit without changing anything in `Utils` and without affecting the output of your interpreter on programs that do not use recursion.

Final Remarks

- ▷ There is a lot of repetition here, this is just the nature of implementing programming languages. So even though there is a lot of code to write, it should go pretty quickly. Despite this, it may be worthwhile to think about how to implement the interpreter without too much code copying.
- ▷ Test along the way. Don't try to write the whole interpreter and test afterwards. We're taking off the training wheels for this project, you will *not* be given any OUnit tests. If you want to test individual functions, you'll have to write your own. We will provide some example programs against which you can test your full interpretation pipeline.
- ▷ You **must** use exactly the same names and types as given at the top of this file. They **must** appear in the file `interp1/lib/interp1.ml`. If you don't do this, we can't grade your submission. You are, of course, allowed to add your own functions.

Good luck, happy coding.