Mini-Project 1: Your First Interpreter

CAS CS 320: Principles of Programming Languages

Due November 7, 2024 by 11:59PM

In this project, you will be building 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. It will be your task to write the following functions (these are the only functions we will tests):

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

The types used in the above signature appear in the module `Utils` and are detailed below. They should all appear in a file `interp01/lib/lib.ml`. **Please read the following instructions completely and carefully.**

Part 1: Parsing

A program in our language is given by the grammar in Figure 1. The start symbol of this grammar is <prog>. We present the operators and their associativity *in order of increasing precedence* in Figure 2. Note that function application is not actually an operator in the language in that it has no associated token the grammar, and so its associativity and precendence will not be specified directly in the parser. That said, it's important to note that function application is left associative because this affects how a sequence of applications should be parsed. See the examples from lecture for more details.

Your implementation of `parse` should return `None` in the case that the input string is not recognized by this grammar. It should target the algebraic data type in Figure 3, which appears in the module `Utils`.

As with OCaml, our language is whitespace agnostic. This means that in your lexer, you should make sure to eliminate whitespace between tokens (you should emulate what was done in the example from lecture). We will use the following regular expressions 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. 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 (as it is 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 (as it is in OCaml). You can used these identifiers without modification in your lexer.

<prog> ::= <expr> EOF
<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*

Figure 1: The grammar for our language

Operators	Associativity
||	right
&&	right
<, <=, >, >=, =, <>	left
+, -	left
*, /, mod	left
function application	left

Figure 2: The operators (and their associativity) of our langauge in order of increasing precedence

```
type prog = expr

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
```

Figure 3: The ADT representing expressions in our language

Part 2: Evaluation

The evaluation of a program in our language is given by the big-step operational semantics presented below. As usual, we write $e \Downarrow v$ to indicate that the expression e evaluates to the value v. For this project, we take a *value* to be:

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

The function `eval`, given an expression e (of type `expr`), should return `Ok` v (of type `(value, error) result`), in the case that $e \Downarrow v$ is derivable according to the given semantics. Otherwise it should return `Error` *err* where *err* is of type `error` (if possible). The error *err* will be 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
- ▷ `UnknownVar`, an unbound variable was evaluated

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.

Operational Semantics

The remainder of this section is a formal description of the big-step operational semantics of our language. Most of these inference rules should feel familiar. Any premise that is not of the form $e \Downarrow v$ is a *side condition*. Note that, since our language has no type checking, our operational semantics is more verbose. We need many side conditions to verify that values are of the correct type for the given conclusion, e.g., we have to check that the operands of an arithmetic operator both evaluate to integers.

With regards to *order of evaluation*, for this course we assume that, given rules for big-step semantics, we evaluate expressions in premises from left to right. This is *important*, as the order of evaluation may affect what error is observed or even whether or not evaluation terminates.

Literals

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

Conditionals

$$\frac{e_1 \Downarrow \mathit{true} \qquad e_2 \Downarrow v}{\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \Downarrow v}\ (\textsf{ifTrueEval}) \qquad \frac{e_1 \Downarrow \mathit{false} \qquad e_3 \Downarrow v}{\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \Downarrow v}\ (\textsf{ifFalseEval})$$

Variables and Functions

These rules depend on *substitution*. You should implement capture-avoiding substitution (as discussed in lecture) in the function `subst`. The expression `subst v x e` should be the result of substituting the value v for x in e, i.e., $[v/x]e$ in mathematical notation. Recall that this will depend on the function `gensym` in `Stdlib320`.

$$\frac{e_1 \Downarrow v_1 \qquad [v_1/x]e_2 \Downarrow v}{\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 \Downarrow v}\ (\textsf{letEval}) \qquad \frac{}{\texttt{fun } x \texttt{ -> } e \Downarrow \lambda x.e}\ (\textsf{funEval}) \qquad \frac{e_1 \Downarrow \lambda x.e \qquad e_2 \Downarrow v_2 \qquad [v_2/x]e \Downarrow v}{e_1\ e_2 \Downarrow v}\ (\textsf{appEval})$$

Arithmetic Operations

The operators '+', '−', '∗', '/' and '*mod*' have their usual definitions as operators on integers. In particular '/' should correspond with integer division in OCaml.

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad v_1 \in \mathbb{Z} \qquad v_2 \in \mathbb{Z}}{e_1 \texttt{ + } e_2 \Downarrow v_1 + v_2} \text{ (addEval)} \qquad \frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad v_1 \in \mathbb{Z} \qquad v_2 \in \mathbb{Z}}{e_1 \texttt{ - } e_2 \Downarrow v_1 - v_2} \text{ (subEval)}$$

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

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad v_1 \in \mathbb{Z} \qquad v_2 \in \mathbb{Z} \qquad v_2 \neq 0}{e_1 \texttt{ mod } e_2 \Downarrow v_1 \ mod \ v_2} \text{ (modEval)}$$

Comparison Operations

$$\frac{e_1 \Downarrow v_1 \qquad e_1 \Downarrow v_2 \qquad v_1 \in \mathbb{Z} \qquad v_2 \in \mathbb{Z} \qquad v_1 < v_2}{e_1 \texttt{ < } e_2 \Downarrow true} \text{ (ltEvalTrue)}$$

$$\frac{e_1 \Downarrow v_1 \qquad e_1 \Downarrow v_2 \qquad v_1 \in \mathbb{Z} \qquad v_2 \in \mathbb{Z} \qquad v_1 \geq v_2}{e_1 \texttt{ < } e_2 \Downarrow false} \text{ (ltEvalFalse)}$$

$$\frac{e_1 \Downarrow v_1 \qquad e_1 \Downarrow v_2 \qquad v_1 \in \mathbb{Z} \qquad v_2 \in \mathbb{Z} \qquad v_1 \leq v_2}{e_1 \texttt{ <= } e_2 \Downarrow true} \text{ (lteEvalTrue)}$$

$$\frac{e_1 \Downarrow v_1 \qquad e_1 \Downarrow v_2 \qquad v_1 \in \mathbb{Z} \qquad v_2 \in \mathbb{Z} \qquad v_1 > v_2}{e_1 \texttt{ <= } e_2 \Downarrow false} \text{ (lteEvalFalse)}$$

$$\frac{e_1 \Downarrow v_1 \qquad e_1 \Downarrow v_2 \qquad v_1 \in \mathbb{Z} \qquad v_2 \in \mathbb{Z} \qquad v_1 > v_2}{e_1 \texttt{ > } e_2 \Downarrow true} \text{ (gtEvalTrue)}$$

$$\frac{e_1 \Downarrow v_1 \qquad e_1 \Downarrow v_2 \qquad v_1 \in \mathbb{Z} \qquad v_2 \in \mathbb{Z} \qquad v_1 \leq v_2}{e_1 \texttt{ > } e_2 \Downarrow false} \text{ (gtEvalFalse)}$$

$$\frac{e_1 \Downarrow v_1 \qquad e_1 \Downarrow v_2 \qquad v_1 \in \mathbb{Z} \qquad v_2 \in \mathbb{Z} \qquad v_1 \geq v_2}{e_1 \texttt{ >= } e_2 \Downarrow true} \text{ (gteEvalTrue)}$$

$$\frac{e_1 \Downarrow v_1 \qquad e_1 \Downarrow v_2 \qquad v_1 \in \mathbb{Z} \qquad v_2 \in \mathbb{Z} \qquad v_1 < v_2}{e_1 \texttt{ >= } e_2 \Downarrow false} \text{ (gteEvalFalse)}$$

$$\frac{e_1 \Downarrow v_1 \qquad e_1 \Downarrow v_2 \qquad v_1 \in \mathbb{Z} \qquad v_2 \in \mathbb{Z} \qquad v_1 = v_2}{e_1 \texttt{ = } e_2 \Downarrow true} \text{ (eqEvalTrue)}$$

$$\frac{e_1 \Downarrow v_1 \qquad e_1 \Downarrow v_2 \qquad v_1 \in \mathbb{Z} \qquad v_2 \in \mathbb{Z} \qquad v_1 \neq v_2}{e_1 \texttt{ = } e_2 \Downarrow false} \text{ (eqEvalFalse)}$$

$$\frac{e_1 \Downarrow v_1 \qquad e_1 \Downarrow v_2 \qquad v_1 \in \mathbb{Z} \qquad v_2 \in \mathbb{Z} \qquad v_1 \neq v_2}{e_1 \texttt{ <> } e_2 \Downarrow true} \text{ (neqEvalTrue)}$$

$$\frac{e_1 \Downarrow v_1 \qquad e_1 \Downarrow v_2 \qquad v_1 \in \mathbb{Z} \qquad v_2 \in \mathbb{Z} \qquad v_1 = v_2}{e_1 \texttt{ <> } e_2 \Downarrow false} \text{ (neqEvalFalse)}$$

Boolean Operations

Note that these operators behave differently from the ones above. These rules describe the kind of *short-circuiting* behavior of Boolean operators as seen in many languages like OCaml.

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

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

Putting Everything Together

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, except that if `parse` fails, `interp` should return `Error ParseFail`. Once you do this, you should be able to run

```
dune exec interp01 filename
```

in order to execute code you've written in other files (replace `filename` with the name of the file which contains code 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)
```

Note that our executable prints out the value of the expression. If you want to compare with the value given by OCamls interpreter, you can use `utop` to see the value (or print it out yourself).

Extra Credit: Recursion

The semantics we've presented here makes it a bit difficult to implement recursion. This is, in part, because, like OCaml, we presume that variable definitions *shadow* existing definitions. 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 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 `interp01/lib/lib.ml`. If you don't do this, we can't grade your submission. You are, of course, allowed to add your own functions and directories (e.g., you will probably want to include a directory which contains your parser generated by Menhir).
- ▷ You are given a skeleton dune project to implement your interpreter. **Do not change any of the given code**. In particular, don't change the dune files or the utility files. When we grade you assignment we will be assume this skeleton code.

Good luck, happy coding.