Graduate Credit Project
CIS705, Kansas State University

Fall 2024

1 General Guidelines

- you may do the project with a partner, or on your own
- you should present your work between December 4th and December 10th (slots to be assigned later), with a presentation lasting 15–20 minutes
- you should submit your code file(s) by Friday, December 13th, including a `README` file on how to run it (if you work in pairs it suffices that one of you submits)
- we will continue to meet at regular class time (starting from November 22th, classes will be canceled for undergraduates).

You may do *either*

1. the project described in Section 2, *or*
2. a project proposed by you that in a significant way involves fundamentals of programming languages as were taught over the course.

   You should (as soon as possible) write a project proposal and submit it to us for approval/modification.

2 The Default Project: Interpreting an OCaml Subset

Goal

Use techniques from the course to implement, from scratch, a non-trivial language.

Language to be Interpreted

is a subset of OCaml.

Types T are given by the syntax

$$
\begin{array}{rcl}
T & ::= & \texttt{int} \\
 & | & I \\
 & | & (T \texttt{ -> } T) \\
 & | & (T \texttt{ * } T)
\end{array}
$$

where I is an identifier, that is a sequence of letters, digits and underscores that starts with a lowercase letter. Such identifiers denote datatypes defined by **declarations** of the form

$$\texttt{type } I \texttt{ = } S$$

where S is a sequence of clauses of the form

$$\texttt{| } C \texttt{ of } T$$

where a **constructor** C is a sequence of letters, digits and underscores that starts with an uppercase letter.

For example, we may have the declaration

```
type tree = | Leaf of int | Node of (tree * tree)
```

Expressions E are given by the syntax

E	$::=$	N	number
	$\mid$	I	identifier
	$\mid$	`(fun` I`:`T `->` E`)`	anonymous function definition
	$\mid$	`(`E E`)`	function application
	$\mid$	`(if` E *relop* E `then` E `else` E`)`	conditional, with *relop* either `<` or `=`
	$\mid$	`(`E *op* E`)`	arithmetic operation, with *op* either `+` or `-` or `*`
	$\mid$	`(`E `,` E`)`	make a pair
	$\mid$	`(fst` E`)`	first component of pair
	$\mid$	`(snd` E`)`	second component of pair
	$\mid$	`(`C E`)`	apply constructor
	$\mid$	`(match` E `with` B`)`	match E with various patterns

where B is a sequence of clauses of the form

$$\texttt{| } C\ I \texttt{ -> } E$$

Let Bindings L are given by the syntax

$$
\begin{array}{rcl}
L & ::= & \texttt{let } I \texttt{ = } E \\
 & | & \texttt{let } I \texttt{ (}I\texttt{:}T\texttt{) = } E \\
 & | & \texttt{let rec } I \texttt{ (}I\texttt{:}T\texttt{) = (}E \texttt{ : } T\texttt{)}
\end{array}
$$

Programs P are formed from

1. a sequence of declarations, followed by
2. a sequence of let bindings (each followed by `;;`), followed by
3. an expression (whose value, given the previous declarations/bindings, is the value of the program).

Remarks Observe that we

- require parentheses several places (some of which optional in OCaml) so as to facilitate parsing
- require explicit type annotations (optional in OCaml) so as to facilitate type checking
- do not allow polymorphism.

Example programs

The program

```
type tree = | Leaf of int | Node of (tree * tree)

let rec add (t : tree) =
   ((match t with
      | Leaf n -> n
      | Node t1 ->  ((add (fst t1)) + (add (snd t1)))) : int) ;;

let leaf1 = 3 ;;
let leaf2 = 4 ;;

(add (Node ((Leaf leaf1), (Leaf leaf2))))
```

should have type `int` and return 7 (which it will do if run in OCaml), and

```
type tree = | Leaf of int | Node of (tree * tree)

let rec map (ft : ((int -> int) * tree)) =
   ((match (snd ft) with
      | Leaf n -> (Leaf ((fst ft) n))
      | Node t0 -> (Node ((map ((fst ft), (fst t0))),
                          (map ((fst ft), (snd t0))))))
     : tree) ;;

(map ((fun n : int -> (n + 1)), (Node ((Leaf 2), (Leaf 4)))))
```

should have type `tree` and return (perhaps with some parentheses inserted)

```
  Node (Leaf 3, Leaf 5)
```

(which it will do if run in OCaml).

Tasks

In a language of your choice (OCaml, Python etc) you must

1. write a parser (which uses a lexical analysis), much as in previous projects
2. write a type checker, which (implicitly) assigns a type to all expressions in the program; the type checker should use
   - a type environment that associates a type to all free identifiers, and also
   - a constructor environment that to each constructor associates its type constraints (in our examples, the constructor `Node` will when applied to a value of type `(tree * tree)` return a value of of type `tree`).
3. write an interpreter, which for a program returns its value, which could be a number, a closure, a pair, or a constructor applied to a value; the interpreter should use a value environment that associates a value to all free identifiers.

Observe that for a program/expression that has passed phase 2, we do not expect errors in phase 3. (As famously stated by Robin Milner: *Well typed programs cannot go wrong.*)

For your inspiration, a skeleton interpreter written in OCaml will be provided, as will various test programs.