CSCI3180 Assignment 3: SAML

2024-2025, Term 2

Introduction

In this two-part assignment, you'll use OCaml to implement an evaluator for an extension of SAML with `let rec`. There are a total of **100 points** for this assignment.

Guidelines

You should follow all below guidelines during completion of the assignment:

1. Modify *only* the files `desugar.ml` and `interpret.ml` by adding your code.
2. You may define additional functions or types if you find it helpful, but ensure that they are given *unique* names so that they don't shadow other values defined in other `desugar.ml` or `interpret.ml`.
3. Do not modify any types, values, or functions already defined in `desugar.ml` or `interpret.ml`.

Failure to adhere to the guidelines may result in a reduction of points from your assignment.

Submission Guidelines

The submission deadline for the assignment is **3 April 2025 at 11:59pm**. There will be a late submission penalty of **1 point per 5-minutes late**, where the amount of time late is rounded *up* to the nearest 5 minutes. You should complete the following by the deadline:

- Ensure that you have completed the declaration at the top of `interpret.ml`.
- On Blackboard, for Assignment 3, attach and submit *only* your completed versions of `desugar.ml` and `interpret.ml`. (Do not zip or otherwise compress the files.) The files *must* be called `desugar.ml` and `interpret.ml`, respectively.

You can submit as many times as you want, but **only the latest submission will be graded**. You are encouraged to submit early to prevent any issues!

Lexing and Parsing

The task of lexing and parsing to convert a string into a SAML AST has already been done for you using the `ocamllex` and `ocamlyacc` parser generators.

The file `lexer.mll` defines the lexer, which converts its input into meaningful *tokens*. From `lexer.mll`, `ocamllex` generated the file `lexer.ml` containing all the code needed to perform lexical analysis.

The file `parser.mly` defines a CFG from which `ocamlyacc` generated the files `parser.ml` and `parser.mli`, the former of which contains the code for converting a sequence of input tokens into a `saml_ast` (defined in `desugar.ml`).

Part 1: Desugaring (30 pts)

In the first part of the assignment, you'll implement a desugarer from SAML to SAMLCORE programs. For this part of the assignment, you only need to modify `desugar.ml`.

Below is the extended grammar for SAML programs (modified so that they now only involve ASCII characters):

$$
\begin{array}{rll}
e \in \langle expr \rangle ::= & \langle id \rangle & \text{identifier} \\
& \mid \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 & \text{if expression} \\
& \mid \texttt{fun } xs \texttt{ -> } e & \text{function definition} \\
& \mid e_1 \ e_2 & \text{function application} \\
& \mid \texttt{let } \langle bind \rangle \texttt{ in } e & \text{let expression} \\
& \mid \texttt{let rec } x \texttt{ = } e_1 \texttt{ in } e_2 & \text{let rec} \\
& \mid \langle op \rangle \ e & \text{unary operation} \\
& \mid e_1 \ \langle binop \rangle \ e_2 & \text{binary operation} \\
\langle bind \rangle ::= & x \texttt{ = } e \mid x \texttt{ = } e \texttt{,} \langle bind \rangle & \\
\langle id \rangle ::= & x \mid z \mid b & \\
xs \in \langle args \rangle ::= & x \mid x \ \langle args \rangle & \\
\langle binop \rangle ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{and} \mid \texttt{or} \mid \texttt{<=} & \\
\langle op \rangle ::= & \texttt{not} &
\end{array}
$$

Below is an ADT defining the type of SAML ASTs:

```
(* Saml AST type *)
type saml_ast =
    Int of int
  | Bool of bool
  | Var of string
  | Let of (saml_ast * saml_ast) list * saml_ast
  | LetRec of saml_ast * saml_ast * saml_ast
  | Ite of saml_ast * saml_ast * saml_ast
  | Fun of saml_ast list * saml_ast
  | App of saml_ast * saml_ast
  | Lte of saml_ast * saml_ast
  | Plus of saml_ast * saml_ast
  | Minus of saml_ast * saml_ast
  | Times of saml_ast * saml_ast
  | Divides of saml_ast * saml_ast
  | And of saml_ast * saml_ast
  | Or of saml_ast * saml_ast
  | Not of saml_ast
```

Meanwhile, below is the extended grammar for SAMLCORE programs (also modified so that they now only involve ASCII characters):

$$
\begin{array}{rll}
c \in \langle expr \rangle ::= & \langle id \rangle & \text{identifier} \\
& \mid \texttt{if } c_1 \texttt{ then } c_2 \texttt{ else } c_3 & \text{if expression} \\
& \mid \texttt{fun } x \texttt{ -> } c & \text{function definition} \\
& \mid \texttt{let rec } x \texttt{ = } c_1 \texttt{ in } c_2 & \text{let rec} \\
& \mid c_1 \ c_2 & \text{function application} \\
& \mid c_1 \langle binop \rangle c_2 & \text{binary operation} \\
\langle id \rangle ::= & x \mid z \mid b & \\
\langle binop \rangle ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{<=} &
\end{array}
$$

Below is an ADT defining the type of SAMLCORE ASTs:

```
(* SamlCore AST type *)
type core_ast =
    CInt of int
  | CBool of bool
  | CVar of string
  | CIte of core_ast * core_ast * core_ast
```

```
  | CLetRec of core_ast * core_ast * core_ast
  | CFun of core_ast * core_ast
  | CApp of core_ast * core_ast
  | CLte of core_ast * core_ast
  | CPlus of core_ast * core_ast
  | CMinus of core_ast * core_ast
  | CTimes of core_ast * core_ast
  | CDivides of core_ast * core_ast
```

Exercise 1 (30 pts)

Define a function `desugar : saml_ast -> core_ast` that desugars SAML expressions in to SAMLCORE expressions. Definitions of SAML expressions as syntactic sugar over SAMLCORE expressions are below:

$$\texttt{fun}\ x\ xs\ \texttt{->}\ e \equiv \texttt{fun}\ x\ \texttt{->}\ \texttt{fun}\ xs\ \texttt{->}\ e$$
$$\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \equiv (\texttt{fun}\ x\ \texttt{->}\ e_2)\ e_1$$
$$\texttt{let}\ x\ \texttt{=}\ e_1, \mathit{binds}\ \texttt{in}\ e_2 \equiv (\texttt{fun}\ x\ \texttt{->}\ \texttt{let}\ \mathit{binds}\ \texttt{in}\ e_2)\ e_1$$
$$e_1\ \texttt{and}\ e_2 \equiv \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else false}$$
$$e_1\ \texttt{or}\ e_2 \equiv \texttt{if}\ e_1\ \texttt{then true else}\ e_2$$
$$\texttt{not}\ e \equiv \texttt{if}\ e\ \texttt{then false else true}$$

Testing

You can build a native executable for testing the desugarer by running the following `ocamlopt -o desugar_tests desugar.ml desugar_tests.ml`. If you run the resulting file (called `desugar_tests` or `desugar_tests.exe`), you should get the following output:

```
desugar integer     : Passed!
desugar variable    : Passed!
desugar binary op   : Passed!
desugar function    : Passed!
desugar function (2): Passed!
desugar let         : Passed!
```

Part 2: Evaluator (70 pts)

In this part of the assignment, you will modify `interpret.ml` to implement a Call-by-Value interpreter for SAMLCORE, following the environment semantics we've seen in class.

Below is the type definition for values, which is defined mutually recursively with the type of environments:

```
type value =
    VInt of int
  | VBool of bool
  | Closure of string * core_ast * env

(* Environment: Maps variable names to values *)
and env = (string * value) list
```

An exception `Stuck` is defined for when the program gets stuck:

```
exception Stuck;;
```

Exercise 2 (70 pts)

Define a function `eval : environment -> core_ast -> value` that implements the environment semantics for SamlCore according to the semantics below:

$$\text{(Int) } E \vdash z \Downarrow z \quad \text{(Bool) } E \vdash b \Downarrow b$$

$$\text{(Closure) } E \vdash \texttt{fun } x \texttt{ -> } e \Downarrow (\texttt{fun } x \texttt{ -> } e, E)$$

$$\text{(App-Closure) } \frac{E \vdash e_1 \Downarrow (\texttt{fun } x \texttt{ -> } e_3, E_c) \qquad E \vdash e_2 \Downarrow v_2 \qquad E_c, x : v_2 \vdash e_3 \Downarrow v_1}{E \vdash e_1\ e_2 \Downarrow v_1}$$

$$\text{(If-True) } \frac{E \vdash e_1 \Downarrow \texttt{true} \qquad E \vdash e_2 \Downarrow v}{E \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \Downarrow v}$$

$$\text{(If-False) } \frac{E \vdash e_1 \Downarrow \texttt{false} \qquad E \vdash e_3 \Downarrow v}{E \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \Downarrow v}$$

$$\text{(Var) } E \vdash x \Downarrow v \text{ if } E(x) = v$$

$$\text{(Lt) } \frac{E \vdash e_1 \Downarrow z_1 \qquad E \vdash e_2 \Downarrow z_2}{E \vdash e_1 \texttt{ <= } e_2 \Downarrow b} \text{ if } b = z_1 \leq z_2$$

$$\text{(Plus) } \frac{E \vdash e_1 \Downarrow z_1 \qquad E \vdash e_2 \Downarrow z_2}{E \vdash e_1 \texttt{ + } e_2 \Downarrow z} \text{ if } z = z_1 + z_2$$

$$\text{(Minus) } \frac{E \vdash e_1 \Downarrow z_1 \qquad E \vdash e_2 \Downarrow z_2}{E \vdash e_1 \texttt{ - } e_2 \Downarrow z} \text{ if } z = z_1 - z_2$$

$$\text{(Times) } \frac{E \vdash e_1 \Downarrow z_1 \qquad E \vdash e_2 \Downarrow z_2}{E \vdash e_1 \texttt{ * } e_2 \Downarrow z} \text{ if } z = z_1 \times z_2$$

$$\text{(Divide) } \frac{E \vdash e_1 \Downarrow z_1 \qquad E \vdash e_2 \Downarrow z_2}{E \vdash e_1 \texttt{ / } e_2 \Downarrow z} \text{ if } z = \left\lfloor \frac{z_1}{z_2} \right\rfloor$$

The semantics for `let rec` expressions is omitted from the above semantics as there are a couple of options for implementing them. It is left up to you to define and implement the semantics for `let rec` functions, *but* the semantics should be defined in such a way that it matches the CBV semantics covered in the course!
For expressions that do not evaluate to a value, `eval` should raise the exception `Stuck`.

Hint: See the notes from class for a refresher on how to implement recursive functions. Of course, the notes on recursion use a substitution-based semantics for Saml. Depending on how you decide to implement `let rec`, you might need to adapt some semantic rules to use environments and closures instead. If you need to do this, it may be helpful to compare the App rule from the substitution-based semantics and App-Closure rule from the environment-based semantics.

Building the Interpreter

The file `evaluator.ml` provides a front-end for the interpreter that uses `desugar` and `eval` to desugar and then evaluate the program contained in the file provided as a command-line argument.

You can build a native executable for the interpreter by running the following:

```
ocamlc -c desugar.ml
ocamlc -c parser.mli
ocamlc -c lexer.ml
```

```
ocamlc -c parser.ml
ocamlc -c interpret.ml
ocamlc -c evaluator.ml
ocamlc -o saml lexer.cmo parser.cmo desugar.cmo interpret.cmo evaluator.cmo
```

Then if you run the resulting file (called `saml` or `saml.exe`) with a file containing a SAML program, you should get the result of evaluating the program as the output.

The files in the `programs` directory are some such example programs.

Below are the expected outputs for these example programs (on a Unix-like system where the executable is called `saml` and is in the current directory):

```
./saml programs/program_1.saml
2
```

```
./saml programs/program_2.saml
120
```

```
./saml programs/program_3.saml
Stuck
```

```
./saml programs/program_4.saml
(Closure: fun x -> x)
```

```
./saml programs/program_5.saml
2
```

The provided Python file `compile_and_evaluate.py` can optionally be used to build the `saml` executable and run `./saml` on all the files in the `programs` directory on a Unix-like system. You may adapt it to suit your needs.

Testing

The programs in the `programs` directory do not include all SAML features, so you are encouraged to write your own SAML programs and test the interpreter on them to make sure everything works!