CS442
Assignment 5

University of Waterloo

Winter 2023

This assignment tests your understanding of the content of Module 7. You will implement the Simple Imperative Language in OCaml. This assignment has three parts: SIL, SIL-P, and SIL-PA.

In this and all assignments, any behavior which we do not explicitly define will not be tested, so you may define it however you wish, or allow your program to fail. Make sure though that it actually *is* undefined; ask on Piazza if you're unsure.

Submit all code via UWaterloo submit, e.g.:

```
submit cs442 a5 .
```

This assignment is due on **THURSDAY**, April 6th, by 12PM *NOON, NOT MIDNIGHT*, Eastern time.

SIL in OCaml

You are provided with a file `SIL.ml`, which implements a parser and some utilities for the Simple Imperative Language, SIL-P, and SIL-PA. Its implementation of SIL-PA is slightly simplified, by restricting array creation (the `array[E]` syntax) only to a specific array-creation statement, `X := array[E]`, so that only statements can add new elements to the heap. In addition, it implements the syntax for a `print` statement, of the form `print E`, where `E` is an expression. Otherwise, it is as described in Module 7. You may use `SIL.ml` as a module either by prefixing its names with `SIL.`, or by using `open SIL` at the beginning of your program. It defines the following types:

```
type expr =
    (* SIL *)
    | Num of int
    | Var of string
    | AddExpr of (expr * expr)
    | MulExpr of (expr * expr)
    | NegExpr of expr
    (* SIL-PA *)
    | ArrIndexExpr of (expr * expr)

type boolExpr =
    | True
    | False
    | Not of boolExpr
    | And of (boolExpr * boolExpr)
    | Or of (boolExpr * boolExpr)
    | Gt of (expr * expr)
    | Lt of (expr * expr)
    | Eq of (expr * expr)

type procedure = string * string list * string list * statement list
```

```
type statement =
    (* SIL *)
    | Skip
    | Block of statement list
    | WhileStmt of (boolExpr * statement)
    | IfStmt of (boolExpr * statement * statement)
    | VarAssgStmt of (string * expr)
    (* SIL-P *)
    | ProcDecl of procedure
    | CallStmt of (string * expr list)
    (* SIL-PA *)
    | NewArrStmt of (string * expr)
    | ArrAssgStmt of (string * expr * expr)
    (* Utility *)
    | PrintStmt of expr

(* A program is a list of statements *)
type program = statement list
```

Most defined types are variant types (i.e., algebraic data types), with each variant being one form. A program is a list of statements. A statement is one of the many statement types in SIL and its extensions:

- `Skip` represents the skip statement.
- `Block sl` represents a `begin`-`end` block, in which `sl` is the statement list.
- `WhileStmt (cond, body)` represents a `while` loop, in which `cond` and `body` are the condition (a boolean expression) and body (a statement) of the while loop, respectively.
- `IfStmt (cond, thenStmt, elseStmt)` represents an `if` statement, in which `cond`, `thenStmt`, and `elseStmt` are the condition, then branch, and else branch, respectively.
- `ProcDecl p` represents a procedure declaration statement declaring procedure `p`. A procedure is, in turn, a four-tuple `nm, params, decls, body`, where `nm` is the string procedure name, `params` is the string list of parameters, `decls` is the string list of variables declared locally to the procedure, and `body` is the procedure body.
- `CallStmt (target, args)` represents a procedure call, calling the procedure named by the string `target`, with the arguments in the expression list `args`.
- `NewArrStmt (target, sz)` represents the modified array creation statement. The string `target` is the variable into which to place the array reference, and `sz` is the *expression* which, when reduced, represents the size of the array to create.
- `ArrAssgStmt (target, idx, expr)` represents an array assignment, in which the string `target` is the variable containing the reference to the array to modify, and `idx` and `expr` are the expressions for the index to modify and the value to put there, respectively.
- `PrintStmt expr` represents the new `print` statement, in which `expr` is the expression which evaluates to the value to print.

The `boolExpr` type represents boolean expressions. `True` and `False` represent the literal true and literal false. `Not b` represents `not b`. `And (l, r)` and `Or (l, r)` represent `l and r` and `l or r`, respectively. `Gt (l, r)`, `Lt (l, r)`, and `Eq (l, r)` represent `l > r`, `l < r`, and `l = r`, respectively.

The `expr` type represents integer (and, in SIL-PA, array) expressions. `Num n` represents the literal number n, an int. `Var v` represents the variable named by `v`. `AddExpr (l, r)` and `MulExpr (l, r)` represent `l + r` and `l * r`, respectively. `NegExpr e` represents negation, i.e, `-e`. `ArrIndexExpr (target, idx)` represents `target[idx]`.

In addition, `SIL.ml` defines the following parsing and utility functions (as well as many others):

```
(* Parse an SIL[-P[A]] program *)
parseSIL : string -> program option

(* "Freshen" names in this procedure. Returns the new parameter names and body. *)
freshenProcedure : string list -> string list -> statement list -> string list * statement list

(* Print a statement list *)
printStatementList : statement list -> ()
```

`parseSIL` parses an SIL-PA program, returning `None` if it doesn't parse. `freshenProcedure`, given a list of parameters, list of declarations, and statement list (body), "freshens" a procedure as described in Module 7, returning a pair of the freshened parameter list and the freshened body. It is not necessary to return the freshened declaration list, because there is no further use for the declarations after they've been used to freshen the variable names. `printStatementList`, and many other `print*` functions, print each of the various types in their SIL form, for debugging purposes.

1 The Simple Imperative Language

Create a file, `a5.ml`, which implements at least the following function:

```
run : statement list -> ()
```

The `run` function runs the Simple Imperative Language program described by its argument, returning nothing. The only visible behavior of `run` comes from `print` statements in the executed SIL program.

The semantics of the Simple Imperative Language are as described in Module 7, and will not be repeated here. The exception, of course, is the `print` statement. When the `print` statement is executed, you should evaluate its expression argument, which must evaluate to an integer, and print that integer, with a newline, to standard output.

For this entire assignment, *you do not need to perform any error checking.* In the case of SIL, if an undefined variable is accessed, you may fail in any way you please; no specific "getting stuck" behavior is required. For this part, if you encounter any SIL-P or SIL-PA statements or expressions, you may ignore them or produce any value (but note that parts 2 and 3 will be implementing SIL-P and SIL-PA). Although error cases will not be tested, infinite loops *will*; if an input program loops forever, `run` should never return.

You should implement all numbers as OCaml `int`s. Don't worry about overflow.

Hints

It is probably possible to implement this imperatively. I considered banning this option, but decided against it; the fact is, implementing this imperatively in OCaml would be more difficult than implementing it functionally, so if you want to make it more difficult on yourself, then by all means do.

Naturally, `run` will need some helper function(s). In particular, you will probably want some kind of "`step`" function which takes a smaller step. Since only a complete `run` function is required, these smaller steps don't necessarily need to precisely follow the formal semantics.

A "real" interpreter for imperative languages would implement loops in the interpreted language with loops in the host language. Unless you're implementing the whole thing imperatively, it will almost certainly be easier to follow the semantics described (i.e., convert the loop into an `if` that contains the loop).

`run` does not take a store as an argument, and indeed, the type and definition of the store is entirely up to you. An efficient implementation would use some kind of hash map, but that's well beyond the efficiency that's needed.

For this part, the store can only contain numbers. Later, the store will be able to contain procedures and variable references. It would be a good idea to design your store flexibly now, so that it doesn't cause you trouble later.

2 SIL-P

Extend the `run` function to implement procedure declarations and procedure calls. `print` does not need to be extended to support printing procedures; it only needs to be able to print integers. Through the SIL-P section of the Module, N was used as a metavariable for numbers, so it's not necessary for you to concern yourself with, e.g., reassigning procedures to different variables at this phase. However, in SIL-PA, N was generalized to all values, and the semantics therefore allowed reassignment of procedures to other variables, so in the next part of this assignment, you will need to treat procedures as values. As such, you're recommended to treat procedures as values here, but not required to.

Hints

A procedure is quite a bulky thing: a string name, two string lists (parameters and declarations), and a statement list for the body. You can use the `SIL.procedure` type as a convenient boxing of these parts.

A "real" interpreter for an imperative language would make a function call for a procedure call, and it's probably possible to implement SIL-P this way. However, the semantics described in Module 7—freshening the procedure's variables and then simply prepending its parameter assignments and body to your statement list—is fairly simple if you're operating over a statement list anyway, particularly since `SIL.freshenProcedure` is provided. Don't forget that OCaml provides `List.append`.

3 SIL-PA

Extend the `run` function to implement the array creation and assignment statements, and the array access expression. Note that when arrays were introduced, we also generalized N to values, and therefore at this stage, you must allow variable reassignment of procedures (e.g., declare a procedure named `foo`, then `bar := foo`). In Module 7, arrays technically may contain any value[1], but you may choose to write arrays to store only integers; variables must be able to store any value. `print` does not need to be extended to support printing arrays; it only needs to be able to print integers.

Hints

Remember that arrays use a label (reference). If I assign an array to a variable `x`, and then `y := x`, any changes I make through `y` should be visible in `x`.

It's critical that the behavior of a program is as defined by SIL-PA. In particular, you may want to consider if there's a way to get the behavior of labels and a heap without explicitly defining a separate Σ in your code. It's possible to get the right semantics with much simpler code.

To create an array filled with zeros,

- If you are using `Base`, use `Array.create ~len:sz 0`, where `sz` is the desired number of elements.
- Otherwise, use `Array.make sz 0`, where `sz` is the desired number of elements.

Examples

A very simple frontend, `frontend.ml`, is provided on the course web site. You can compile your code with this frontend like so:

```
$ ocamlfind ocamlc -package core -linkpkg -thread SIL.ml a5.ml frontend.ml -o frontend
```

[1] It's a bit ambiguous because of the order in which we introduce things, but N is redefined, and that redefinition should be taken retroactively.

It takes two command-line arguments: a program to run, and an integer value to assign to the `arg` variable. Programs designed to be run with this frontend use the global `arg` variable to control their behavior. You don't need to worry about somehow externally defining `arg`; the frontend does it by prepending a normal variable assignment statement to the parsed program.

Several example programs are provided on the course web site, each of which expect an `arg` variable. You should read them to understand what they do. This is what they should look like in action with a few inputs:

```
$ ./frontend facimp.sil 1
1
$ ./frontend facimp.sil 10
3628800
$ ./frontend fibimp.sil 3
2
$ ./frontend fibimp.sil 20
6765
$ ./frontend facproc.sil 1
1
$ ./frontend facproc.sil 10
3628800
$ ./frontend factorize.sil 2
2
$ ./frontend factorize.sil 14
2
7
$ ./frontend factorize.sil 1234
2
617
$ ./frontend factorize.sil 1235
5
13
19
$ ./frontend fibarr.sil 10
1
1
2
3
5
8
13
21
34
55
```

Rights