First Weekly Assignment for the IPS Course

This is the text of the first weekly assignment for the DIKU course "Implementation of Programming Languages (IPS)", 2023.

Hand in your solution in the form of a short report in text or PDF format, along with whatever extra files (source code and tests) are mandated by the various subtasks. Collect the latter together in a single ZIP file, possibly with subdirectories as relevant. **Do not put your report, nor any binaries, in the ZIP file!**

Task 1: A couple of lines about yourself

In order to adapt our teaching (methods) to the specific audience of this course, we would like you to provide a bit of useful information about yourself.

1. Are you a DIKU Computer Science BSc student (general profile, i.e., with IPS compulsory), or do you come from another line of study?
2. What is your self-perceived level of proficiency in functional programming with F# (e.g. do you remember the curriculum from "Programming and problem solving (PoP)")?
3. How familiar are you with assembly programming (can you write small assembler programs or do you need some brushing up)? And in particular, have you had prior experience with the MIPS and/or RISC-V architectures?
4. What are your expectations regarding this course (i.e., what topics do you find most interesting)?

Task 2: Getting Reacquainted with F#; Using AbSyn and Symbol Tables.

The intent of this task is to help you get reacquainted with programming in F#, and to gain practical experience working with abstract syntax trees (AbSyn or AST) and symbol tables.

Your task is to finish the implementation of an interpreter for a small, but nontrivial integer-calculator language. This is done by completing the implementation of the `eval` function in the handed-out file `calculator.fsx`, which you will then hand back in.

The language covered by the interpreter has the following syntax:

```
Exp ::= n
      | x
      | Exp `+` Exp
      | Exp `-` Exp
      | Exp `*` Exp
      | `(` Exp `)`
      | `let` x `=` Exp `in` Exp
      | `sum` x `=` Exp `to` Exp `of` Exp
      | `prod` x `=` Exp `to` Exp `of` Exp
      | `max` x `=` Exp `to` Exp `of` Exp
      | `argmax` x `=` Exp `to` Exp `of` Exp
```

Here, `n` represents an (unsigned) integer constant, and `x` a variable name (one or more lowercase ASCII letters and/or digits, but starting with a letter, and different from all the keywords). Keywords and symbols (i.e., the terminals of the grammar) are written between quotes in the grammar above for extra emphasis, but the quotes are not part of the actual input. For example, `let v0 = 1+2 in v0*v0` could be a possible expression. Parentheses can be used to override the usual precedences and associativities of the arithmetic operators in the parser, but play no role in the interpreter.

The intended meanings of all but the 4 last expression forms are the usual ones. The expression `sum x = e1 to e2 of e3` corresponds to a mathematical summation (∑), where we compute the sum of the values of `e3` as `x` ranges over the values of `e1` through `e2`. (The bounds `e1` and `e2` are evaluated only once, before the summation starts.) For example, `sum i = 1 to 4 of i*i` should evaluate to 1×1 + 2×2 + 3×3 + 4×4 = 30. If the summation range is empty (i.e., if the upper bound is strictly less than the lower bound), the result is 0 by common convention. Analogously, the `prod`-form computes the product (∏) of the values of the body expression `e3` when `x` ranges from `e1` to `e2`, and returns 1 for an empty range.

Similarly, `max` computes the maximum value of the body over the given integer range; however, unlike the previous two forms, max-ing over an empty range is *undefined*, and should result in an error message. Finally `argmax` is similar to `max`, but instead of returning the maximal value of `e3`, it returns the value of `x` (from the range) for which the maximum value of `e3` is attained. If several different values for `x` result in the same maximal value, the *least* such x is returned.

For example, `max x = 0 to 10 of 5*x-x*x` should return 6, and `argmax x = 0 to 10 of 5*x-x*x` should return 2. (Note that, when considered as function over the real numbers, f(x) = 5×x – x^2 actually attains its maximum for x = 2.5, with f(2.5) = 6.25, while f(2) = f(3) = 6, but the calculator only works with integers.

The abstract syntax of expressions is similar to the one in the slides for the Interpretation lecture, but with a bit deeper structure:

```
type VALUE = INT of int

type BINOP = BPLUS | BMINUS | BTIMES

type RANGEOP = RSUM | RPROD | RMAX | RARGMAX

type EXP =
  | CONSTANT  of VALUE
  | VARIABLE  of string
  | OPERATE   of BINOP * EXP * EXP
  | LET_IN    of string * EXP * EXP
  | OVER      of RANGEOP * string * EXP * EXP * EXP
```

Note that there is only one node type for the three binary operations, with the specific operation determined by an extra argument to the constructor. Similarly, the four range-based operations share a single EXP constructor. For example, the expression `sum x = 1 to 4 of x*x` is represented as

```
OVER (RSUM, "x", CONSTANT (INT 1), CONSTANT (INT 4),
      OPERATE (BTIMES, VARIABLE "x", VARIABLE "x"))
```

We specify that an occurrence of a variable in an expression always refers to the *innermost enclosing* binding for that variable, whether by a `LET_IN` or an `OVER`. That is, inner variable bindings can temporarily *shadow* outer ones.

You are to implement the evaluation of expressions, by filling in the `let rec eval (vtab : SymTab) (e : EXP) : VALUE` function. Currently, only the case for constants is handled.

For convenience, the handed-out code already provides ad-hoc lexing and parsing functionality, together with an interactive environment. Assuming a correct implementation of the `eval` function, the compilation and use of the program is demonstrated below. (Of course, with the handed-out code, only the interpretation of `4` will succeed; the rest will fail with a message "Case for XXX not handled".)

```
$ dotnet fsi calculator.fsx
Welcome to the calculator! Type "exit" to stop.
Input an expression : 4
Evaluation result   : INT 4
Input an expression : 1 - 2 - 3
Evaluation result   : INT -4
Input an expression : let x = 4 in x + 3
Evaluation result   : INT 7
Input an expression : let x0=2 in let x1=x0*x0 in let x2=x1*x1 in x2*x2
Evaluation result   : INT 256
Input an expression : prod x = 1 to 5 of x
Evaluation result   : INT 120
Input an expression : exit
$
```

You should hand in the entire modified `calculator.fsx` file, ready to run. *Additionally*, in a short report, you should explicitly *self-assess the quality of your code*, according to the following parameters:

1. Completeness. Is all the asked-for functionality implemented, at least in principle, even if it's not necessarily fully working? If not, do you have any concrete ideas for how to implement the missing parts?
2. Correctness. Does all implemented functionality work correctly, or are there known bugs or other limitations? In the latter case, do you have any ideas on how to potentially address those problems?
3. Efficiency. Does the *time* and (especially) *space* usage of your code (as you would expect it to be executed by F#; you don't need to actually benchmark it) reasonably match, at least asymptotically, what one would expect for an evaluator of mathematical expressions with summation, etc.? If not, do you have ideas on how to non-trivially reduce the resource usage of your code?
4. Code sharing/elegance. Are common code snippets reasonably shared through parameterized auxiliary definitions, or is there a lot of code duplication in the form of copy-pasted segments with minor changes? Any other ugliness/beauty?
5. Anything else you consider relevant.

The first two points should be briefly substantiated by demonstrating the behavior of your code on a range of relevant input expressions. You may use the provided interactive loop for this, in the style as above. (But don't just throw up a large expression and its output without comment, if it's not immediately obvious what the expected result should be.) For the others, you should also justify your assessment by suitable examples or other evidence.

Task 3: A First Non-Trivial Fasto Program

The intent of this task is to compel you to install and run your first (very simple) Fasto program as soon as possible (i.e., no later than the first week of the course — see group-project document). Note that this task can be completed using just the handed-out, incomplete version of the compiler.

- The Fasto compiler does not yet have multiplication implemented, only addition and subtraction. Therefore, the first subtask will be to develop an auxiliary Fasto function `fun int mul(int x, int y) = …` for multiplying two integers by repeated addition. This function must work correctly even when one or both of `x` and `y` are negative, but it is *not* expected to be efficient. (Hint: you may want to exploit that Fasto functions can be recursive.) Test your function thoroughly, to make sure it works, but you don't need to separately document this testing. Include your declaration of `mul` in the following program.
- The `main` function of your Fasto program, in the file `assign1.fo`, should read in a positive integer `n`. If this `n` is less than or equal to `0`, then the program should write an "Incorrect Input!" message, and terminate immediately (with the integer result `0`).
- Otherwise (i.e., if `n > 0`), the program should create an array of integers (named `arr`), of length `n`, by reading its elements from the standard input. (*Hint:* this may be achieved by mapping the `iota(n)` array by a function that reads an integer; for example, the latter can be declared as `fun int readInt(int i) = read(int)`.)

- Next, a `map` operation should compute the differences `difs` between the consecutive elements of `arr`. That is, `difs` should be another array of length `n`, with its elements determined as follows:
  - `difs[0]` should be `arr[0]`, and
  - `difs[i]` should be `arr[i] - arr[i-1]`, for all `0 < i < n`
- Finally, the result of the program (on this branch) should be the sum of the *squares* of the elements of the `difs` array. (Hint: use a `reduce` here, in addition to your `mul` from above). This result should also be written to standard output.

After writing the program:

- Test your program in both interpretation mode (option `-i`) and compilation mode (option `-c` + using RARS to simulate the execution of generated RISC-V code).
- Then place the `assign1.fo` program in Fasto's `tests/` subdirectory (or a separate one), together with reference input (`assign1.in`) and output (`assign1.out`) files, and validate them with the `runtests.sh` tool. You are encouraged to include additional test cases as well.

For example, suppose the input file (`assign1.in`) contains the following:

```
4
3
7
2
4
```

This corresponds to `n=4` and the elements of `arr` being `{3,7,2,4}`. Then the expected final result is $(3-0)^2 + (7-3)^2 + (2-7)^2 + (4-2)^2 = 9 + 16 + 25 + 4 = 54$. and so `assign1.out` should contain `54`.

In the report, briefly summarize any non-obvious implementation choices in your Fasto program, and the results of your testing. Submit your `assign1.fo` source code, as well as any additional test cases (inputs and corresponding expected outputs) you developed beyond the mandatory one above.