CSL765: Introduction to (Logic and Functional) Programming
I semester 2019-20

The Expression language FL(X)

The document FL(X) describes an expression language The language consists constructors and variables over two datatypes - integer and booleans. The set of terms $T_\Sigma(X)$ is defined as follow-

$$t ::= x \in X \mid Z \mid (P\ t) \mid (S\ t) \mid T \mid F \mid (ITE\ \langle t, t_1, t_0 \rangle) \mid (IZ\ t) \mid (GTZ\ t)$$

Here, X is an infinite set of variables, while Z, P, S, T, F, ITE, IZ, and GTZ are constructors disjoint from the set X. The language makes sure that all the expressions in the language have a unique normal form.

Problem Statement

Your task is to generate a normal form for any given input in language **FL(X)**. Make sure your implementation is complete and efficient since **a part of or the entire assignment might be used in upcoming assignments**. We have attached a signature file with this document. Your task is to create a module and implement the signature file named `signatureFLX.sml`.

What you have to do

1. Create a file `structureFLX.sml` implementing the signature provided given in `signatureFLX.sml`.
2. Name your structure that implements the given signature `Flx`. We will use `open Flx` in our script to use your structure.
3. Write a parser and tokenizer to process the input string keeping in mind the following.
   - The name of the constructors are the same as those provided in the language signature viz. Z, P, T, F, ITE, IZ, and GTZ
   - Every constructor will be followed by at least one space character.
   - The form of application of each constructor P, S, IZ, and GTZ will be `‘(’ <cons name> ‘ ’ <term> ‘)’`
   - The form of application of constructor ITE will be `‘(’ ITE ‘ ’ ‘<’ <term>‘,’ <term1>‘,’ <term0> ‘>’ ‘)’`.
   - You can safely assume any token apart from constructors, parentheses, comma and angular brackets to be the name of a variable i.e, a string $\in X$.
4. Check if the given input is a well formed expression in the language **FL(X)**.
5. The function `fromString: string -> term` should take a string as input and return the term created after parsing. The function should raise `exception Not_wellformed` if the input string is not well formed in the language **FL(X)**.
6. Implement the function `toString: term -> string` which returns a string form of input term.
7. Function `fromInt: int -> term` takes an integer and converts it to an $SS \ldots SZ$ or $PP \ldots PZ$ form in the datatype term. For example `fromInt 2` should return $(S\ (S\ Z))$.

8. Function `toInt: term -> int` should take an integer term in *normal form* and return the corresponding integer number e.g, `toInt (S (S Z))` should return `2` and `toInt (P (P Z))` should return `~2`. If the term consists of a mix of `S` and `P` raise the exception `not_nf`. If the input term consists of any other constructor, raise `not_int` exception.

9. Implement the reduction rules to create a normal form of any well-formed expression.

10. Create a function `normalize: term -> term`, which produces a normal form of the input term as output.

11. You are **not supposed to handle any exception in your program**. Our script depends on you raising the exception from your code.

12. Do not write any code in the signature file `signatureFLX.sml` attached with the assignment.

Submission Instruction

Submit a `structureFLX.sml` file on Moodle `https://moodle.iitd.ac.in`

Important Notes

1. Do not change any of the names given in the signature. You are not even allowed to change upper-case letters to lower-case letters or vice-versa.

2. You may define any new functions you like besides those mentioned in the signature.

3. Follow the input output specification as given. We will be using automated scripts to execute the code for evaluation. In case of mismatch, you will be awarded zero marks.

4. All of your code should be in file `structureFLX.sml`

5. Make sure your structure matches with the signature provided. We will not entertain any requests regarding minor change in the signature.

6. Your functions in this assignment **might be used as an argument to higher order functions and type inferencing in upcoming assignments**. Make sure to write efficient code.