Assignment 3: Type checker

Deadline: 24 november 2021, 23:59

1 Introduction

In this assignment, you will write a program that type checks expressions in the simply-typed lambda calculus. The grammar used is almost the same as in the first assignment, and specified below.

The assignment submission must include a program that:

- reads a judgement from a file into a character string
- lexically analyzes and parses the judgement into an abstract syntax tree
- checks whether the judgement is derivable in the type system

The program must be able to detect syntax errors and should report an error if detected. The program must not make use of external libraries. The program should use the least amount of standard library code. The assignment submission must include a Makefile that can be used to compile the program (except for the Go language: `go build` must work).

The assignment submission must not include any binaries, and must include a README file that documents:

- The class and group number, and the names of the student(s) who worked on the assignment. (Putting the names of the student(s) in each source file is good practice too.)
- The compiler version and operating system used by the student(s).
- Whether it is known that the program works correctly, or whether the program has known defects.
- Whether there are any deviations from the assignment, and reasons why.

The README may include an explanation of how the program works, and remarks for improving the assignment. Finally, the assignment submission may include the following two files:

- An archive of the positive examples used for testing
- An archive of the negative examples used for testing

2 Interface

The program must be compilable and work on the command line. It accepts one command line argument, namely the file from which it reads.

Input. The program reads a string of characters from the file named by the first command line argument. It is permissible that the program only works for files which contain only printable ASCII characters and whitespace. The program accepts one judgement in the input file, but may process multiple judgements, one per line.

Process. After parsing the input into an abstract syntax tree, the program performs type checking. The type checking process must halt.

Exit status. The program must exit with exit status 0 if the judgement is derivable. The program must exist with exit status 1 whenever there is a syntax error, or the judgement is not derivable.

Output. If the program exists with exit status 0 then the program should output the judgement printed to standard output in an unambiguous and standardized output format (where each complex expression and type is surrounded by parentheses). It is permissible that the program outputs intermediary abstract syntax trees on standard error, using a special debugging constant in the program to enable/disable such verbose, diagnostic output. If the program exists with exit status 1 then an error message may be printed to standard error. The program may print understandable error messages.

3 Grammar

The input file is analyzed using the following Backus-Naur grammar:

⟨judgement⟩ ::= ⟨expr⟩ `':'` ⟨type⟩

⟨expr⟩ ::= ⟨lvar⟩ | `'('` ⟨expr⟩ `')'` | `'\'` ⟨lvar⟩ `'^'` ⟨type⟩ ⟨expr⟩ | ⟨expr⟩ ⟨expr⟩

⟨type⟩ ::= ⟨uvar⟩ | `'('` ⟨type⟩ `')'` | ⟨type⟩ `'->'` ⟨type⟩

where ⟨lvar⟩ stands for any variable name that starts with a lowercase letter, and ⟨uvar⟩ stands for any variable name that starts with an uppercase letter. A variable name is alphanumerical: it consists of the letters a-z, A-Z, or the digits 0-9. The grammar should be whitespace insensitive, but whitespace must be recognized to separate application of two variables. The program may support international variable names (i.e. Unicode), and also accept λ instead of \.

The program must support using parentheses in the input to disambiguate expressions and types. If no parentheses are used, the order of presedence for the expression operators is as follows: lambda abstraction groups more strongly than application (i.e. abstraction precedes application), and application associates to the left. The function type constructor associates to the right. The program may support a dot after the lambda abstraction type, where the dot is parsed in the same way as if an opening parenthesis was inserted with a matching closing parenthesis before the next unmatched closing parenthesis or the end of the expression.

3.1 Positive examples

The following examples show input judgements (in a standard output format):

- `(\x^A (\y^(A->B) (y ((\x^A x) x)))):(A -> ((A -> B) -> B))`
- `(\x^A x):(A -> A)`
- `(\x^B (\x^A x)):(B -> (A -> A))`
- `(\y^A (\x^(A -> (C -> A)) (x y))):C -> A`
- etc.

3.2 Negative examples

Invalid input causes the program to terminate with exit status 1:

- `(x y):A` the variables `x` and `y` have an unknown type
- `(\x x)`: missing type
- etc.

4 Evaluation criteria

The submission will be evaluated on the following criteria:

- Correctness of the program (hard criterium, 60%): is the program correctly implementing the assignment? Are there cases in which the program is implemented incorrectly?
- Readability of the program (soft criterium, 30%): is the program written to be understandable to humans too?
- Efficiency of the program (soft criterium, 10%): is program executing without noticable delay?

In the above text, the words must, should, and may have a special meaning. The assignment is graded with a passing grade if all features that must be implemented are correctly implemented. Higher grades are for submissions that also correctly implement features that should be implemented. Even higher grades are for submissions that also correctly implement features that may be implemented.