Assignment 2: Interpreter

Deadline: November 9th, 2022, 23:59

1 Introduction

In this assignment, you will write a program that interprets expressions in the lambda calculus. The grammar used is the same as the previous assignment, and specified below.

The assignment submission must include a program that:

- reads an expression from a file into a character string
- lexically analyzes and parses the expression into an abstract syntax tree
- performs reductions on the abstract syntax tree, until no longer possible
- outputs a character string corresponding to the final abstract syntax tree

The program must be able to detect syntax errors and then should report an error. The program may terminate with an error after a fixed amount of reduction steps (e.g. 1000) if then there are still reductions possible. 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 (if applicable).

The assignment submission 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) if applicable.
- 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 (`positive.tar.gz`) of the positive examples used for testing.
- An archive (`negative.tar.gz`) of the negative examples used for testing.

2 Interface

The program must be compilable (if applicable) and work on the command line. The program can then be invoked using the command line. It accepts one required 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. The program must work for files which contain only printable ASCII characters and whitespace but may also work for files which contain non-printable ASCII characters. The program should accept only one expression in the input file.

Process. After parsing the input into an abstract syntax tree, the program performs α-conversions and β-reductions. The α-conversions should only be performed if a β-reduction would otherwise lead to a captured variable. If there are multiple places in an expression where a β-reduction can be performed, the program chooses an arbitrary place where reduction is performed. The reduction strategy may be configured, e.g. using command line flags. The README must document what reduction strategies are implemented.

Exit status. The program must exit with exit status 0 whenever the expression cannot be reduced any further by a β-reduction. The program must exist with exit status 1 whenever there is a syntax error, or not enough command line arguments are supplied. The program may exit with exit status 2 whenever a limit on the number of reduction steps has been reached. Alternatively, if the program runs forever on some inputs it must be interruptable by the operating system.

Output. If the program exists with exit status 0 then the program must have outputed the reduced abstract syntax tree, in a standard format, to the standard output. 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 or exit status 2 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:

$$\langle\text{expr}\rangle ::= \langle\text{var}\rangle \mid \texttt{'('}\ \langle\text{expr}\rangle\ \texttt{')'} \mid \texttt{'\textbackslash'}\ \langle\text{var}\rangle\ \langle\text{expr}\rangle \mid \langle\text{expr}\rangle\ \langle\text{expr}\rangle$$

where ⟨var⟩ stands for any variable name. A variable name is alphanumerical: it consists of the letters a-z, A-Z, or the digits 0-9. A variable name must start with a letter from the alphabet, i.e. not with a digit. 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. If no parentheses are used, the order of presedence for the operators is as follows: lambda abstraction groups more strongly than application (i.e. abstraction precedes application), and application associates to the left. The program may support a dot after the lambda abstraction variable, where the dot is parsed in the same way as if an opening parenthesis was inserted in the place of the dot with a matching closing parenthesis extending as much as possible to the right.

3.1 Positive examples

The following examples show input expressions and their reductions:

- `(x y)` contains no place where β-reduction can be performed, so this expression is the output
- `\x\y(x\z y)` contains no place where β-reduction can be performed
- `(\x x)(\y y)` immediately β-reduces to `(\y y)` which is then the output
- `(\x \y x)(\z y)` first performs α-conversion to `(\x \w x)(\z y)` and then β-reduces to `(\w \z y)` which is then the output
- `(\x y)((\x (x x))(\x (x x)))` performs β-reduction and outputs `y` (N.B. compare with the negative example)
- `(\x x x)(\x x x)` β-reduces to `(x)(\x x x)` or `(\x x x)(x)` and then β-reduces to `(x x)`
- etc.

3.2 Negative examples

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

- `(\x`
- `x x))`
- etc.

The following examples cause the program not to terminate or output with exit status 2 if it reaches a limit:

- `(\x (x x))(\x (x x))` β-reduces to `(\x (x x))(\x (x x))` etc. etc.
- `(\x y)((\x (x x))(\x (x x)))` keeps performing β-reduction in the right-hand side of the outer application (N.B. this is also valid behavior)
- 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.