Assignment 1: Lexing and parsing

Deadline: 29 september 2021, 23:59

1 Introduction

In this assignment, you will write a program that lexically analyses and parses a piece of text. The grammar used is that of the lambda calculus, as explained during the lectures, and specified below.

The assignment submission must include a program that:

- reads an expression from a file into a character string
- lexically analyzes the character string into a string of tokens
- parses the token string using recursive descent
- outputs a character string in standard format

The program must be able to detect syntax errors and then report an error. 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.

The assignment submission must includes a README file that documents:

- The class and group number, and the names of the student(s) who worked on the assignment.
- 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. The program can then be invoked using 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. The program may only work for files which contain only printable ASCII characters and whitespace.

Exit status The program must exit with exit status 0 whenever the file was parsed correctly. The program must exist with exit status 1 whenever there was a syntax error, or not enough command line arguments are supplied.

Output If the program exists with exit status 0 then the program must have outputed the input, in a standard format, to the standard 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:

⟨expr⟩ ::= ⟨var⟩ | `'('` ⟨expr⟩ `')'` | `'\'` ⟨var⟩ `'.'` ⟨expr⟩ | ⟨expr⟩ ⟨expr⟩

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 can be used to separate application of two variables. The program may support international variable names (i.e. Unicode), and 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 binds more strongly than application, and application associates to the left.

If parsing is succesful, the output of the program must be again acceptable by the program to parse: the program then succesfully parses its own output and should produce the exact same result. The output should be an unambiguous expression, i.e. with sufficiently many parentheses inserted so the parser never applies any of the presedence rules. The output may use the least amount of whitespace and parentheses in its output.

3.1 Positive examples

The following examples are acceptable:

```
(a b)
abc
a b c
a (b c)
(\ x . a b)
(\x. ((a) (b)))
```

The following outputs are correct (there are also other correct outputs):

```
(a)b
abc
((a)b)c
a((b)c)
(\x.a)b
\x.((a)b)
```

3.2 Negative examples

The following examples are not acceptable:

```
\x
((x
()
a (b
a (b c))
```

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.