universität innsbruck

Advanced Functional Programming	WS 2026/2027	LVA 703139
Exercise Sheet 7, 10 points		Deadline: Tuesday, November 24, 2026, 4pm

- Solve the tasks in files `Exercise07.hs` and upload only this file in OLAT.
- Mark the solved exercises in OLAT.
- Your modified `Exercise07.hs` file must compile with `ghci` without error messages.

The whole exercise sheet is about writing a parser for the ARI format. The format is described in detail at

https://project-coco.uibk.ac.at/ARI/

We restrict ourselves to the TRS format of the ARI format.

https://project-coco.uibk.ac.at/ARI/trs.php

An example TRS is given here:

```
; @author Takahito Aoto
; @author Yoshihito Toyama
; @cops 157
; full experiments for [35]
(format TRS)
(fun * 2)
(fun + 2)
(fun 0 0)
(fun s 1)
(rule (+ x 0) x)
(rule (+ x (s y)) (s (+ x y)))
(rule (* x 0) 0)
(rule (* x (s y)) (+ (* x y) x))
(rule (+ (+ x y) z) (+ x (+ y z)))
(rule (+ x y) (+ y x))
```

An ARI-TRS-file is split into a header (every line starting with ;) and a content part. The latter consists of the format identifier (`format TRS`), the signature describing the arities of the function symbols (`fun fName arity`), and the rules, where each rule consists of two terms which are formed by the following informal grammar:

```
term -> identifier | (identifier term ... term)
```

So, constants and variables are not put into parenthesis, and function application `f(t1,...,tn)` is written as `(f t1 ... tn)` if `n` is larger than zero. There are no infix operators, no precedences, etc. In order to distinguish variables from constants, one has to look into the signature: every identifier that is not a function symbol from the signature can be used as variable.
Note that only comments are restricted to one line. Both signature and rules might be written in a single line, or split over several lines, e.g.,

```
(format
  TRS)(fun * 2) (   fun   +  2  )  (fun 0
  0)(fun s

1)
(rule
  (+ x 0)
  x
) (rule ... remaining rules from previous example ...)
```

is also a valid ARI-TRS-file.

Task 1 *Parsing the Header* **3 p.**

Develop a parser for the header. The header distinguishes arbitrary comments and metadata. Metadata are special comments that start with `"; @"`. They resemble key-value pairs, where the key is some arbitrary string not containing white space and the value is a string not containing newlines. In a meta-info line the key is written using a leading @ symbol, separated from the value by a single space. For example, in the first line of the example TRS the key is "author" and the value is "Takahito Aoto".
The parser for the header should return the metadata.

Task 2 *A Simplified ARI Parser* **4 p.**

Develop a simplified parser for then content part of ARI-files, i.e., the TRSs. The simplified parser may ignore the signature and the metadata and should just parse the rules with the simplifying assumption that all separate identifiers are variables and all function applications are always of the form `(f term term ... term)`.
According to the grammar, identifiers are non-empty strings that do not contain blank, `\t`, `\n`, `(`, `)`, `;`, and `:`.
Note that we use a simplified version of the ARI format compared to the official one from the website: the official version also allows comments outside the header and the identifiers are a bit more restricted.
Example: The first rule of the example TRS should be parsed as:
`(Fun "+" [Var "x",Var "0"],Var "x")`
Once, you solved Task 1 and Task 2 you should be able to execute `cabal run` and get no failures. Here, all TRSs from the ARI-database are parsed.
You can also do quick tests in ghci, e.g., by running
`parse content "" "(format TRS)(rule (f x) x)"`

Task 3 *Taking Signatures into Account* **3 p.**

Extend your parser so that it takes the signature into account, i.e., constants and variables must be distinguished and function applications must have the correct arity, i.e., in the above example all of the following terms are illegal.

```
s             -- s has arity 1 and is not a constant
(+ 0 x y)     -- + has arity 2
(+ x)         -- + has arity 2
```

The first rule of the example TRS must now be parsed as:
`(Fun "+" [Var "x",Fun "0" []],Var "x")`
The checks on the arities should be done throughout the parsing, and not in a post-processing step. The reason is that then failures are reported with positions from the parser.
After completion, you should still not get any errors when running `cabal run`.