Homework 12: A Parser for a Little Language

CIS 352: Programming Languages

13 April 2018, Draft

Administrivia

- Trade ideas with another student? Note it in your cover sheet.
- Turn in, via Blackboard: *(i)* the source files, *(ii)* the transcripts of test runs, and *(iii)* your cover sheet.
- Let me know if any of my QuickCheck tests seem dodgy.

Grading Criteria

- The homework is out of 100 points.
- Omitting your name(s) in the source code looses you 5 points.

Background

In this assignment you will build a parser for a particular syntax for first-order predicate logic. Here is the grammar, which will win no beauty prizes.

⟨*Lower*⟩	::= `a` | `b` | ... | `w` | `x` | `y` | `z`	*[all lowercase letters]*
⟨*Notx*⟩	::= `a` | `b` | ... | `w` | `y` | `z`	*[all lower case letters but x]*
⟨*Upper*⟩	::= `A` | `B` | ... | `Z`	*[all upper case letters]*
⟨*Const*⟩	::= ⟨*Notx*⟩⟨*Lower*⟩*	*[constants]*
⟨*Var*⟩	::= `x0` | `x1` | ···	*[variables]*
⟨*Indiv*⟩	::= ⟨*Var*⟩ | ⟨*Const*⟩	*[individuals]*
⟨*RelName*⟩	::= ⟨*Upper*⟩ ⟨*Lower*⟩*	*[relation names]*
⟨*RelExp*⟩	::= ⟨*RelName*⟩[⟨*Indiv*⟩{, ⟨*Indiv*⟩}*]	*[relation expressions]*
⟨*Expr*⟩	::= ⟨*Term*⟩{ + ⟨*Term*⟩}*	*[expressions]*
⟨*Term*⟩	::= ⟨*Factor*⟩{ & ⟨*Factor*⟩}*	*[terms]*
⟨*Factor*⟩	::= ~⟨*Atom*⟩ | ⟨*QuantExp*⟩	*[factors]*
⟨*QuantExp*⟩	::= `Forall`⟨*Var*⟩(⟨*Expr*⟩) | `Exists`⟨*Var*⟩(⟨*Expr*⟩)	*[quantified expression]*
⟨*Atom*⟩	::= ⟨*RelExp*⟩ | (⟨*Expr*⟩)	*[atomic expressions]*

Notes:

- *Constants* (e.g., `koko`, `wukong`, and `turnip`) are names of particular people and things.
- *Variables* (e.g., `x0`, `x17`, and `x384`) range over people and things.
- *Individuals* are names of people and things, either particular ones (constants) or unspecified ones (variables).
- *Relation names* (e.g. `Likes` and `Tolerates`) are names of some relation between individuals.
- *Relation expressions* are of the form ⟨*RelName*⟩$[a_1, \ldots, a_k]$ where each a_i is either a variable or a constant. E.g.:

  - ➤ `Monkey[wukong]`
  - ➤ `Likes[wukong,koko]`
  - ➤ `Tolerates[mightyjoe,x12]`
- Quantified expressions are of the form: `Forall`⟨*Var*⟩(⟨*Expr*⟩) or: `Exists`⟨*Var*⟩(⟨*Expr*⟩). E.g.:
  - ➤ `Forall x1 (Likes[x1,koko])`
  - ➤ `Exists x0 (Forall x1 (Tolerates[x1,x0]))`
- ~ is symbol for *negation*.
- + is the symbol for *disjunction* (a.k.a. *or*).
- & is the symbol for *conjunction* (a.k.a. *and*).
- Both + and & are left-assocative.
- *precedence*(+) < *precedence*(&) < *precedence*(~).
- More examples:
  - ➤ `Likes[koko,wukong] & Likes[wukong,koko]`
  - ➤ `~(Likes[mightyjoe,wukong] + Tolerates[mightyjoe,wukong])`
  - ➤
```
(Exists x1 (Smaller[x1,mightyjoe]))
    + (Forall x2 (~Smaller[x2,mightyjoe]))
```

Files for this assignment

`LL.hs`	has the abstract syntax for the language (plus printing functions and QuickCheck generators).
`LParser.hs`	has the beginnings of a parser for the language. Do all your work in this file.

I filled in library imports and the parsers for ⟨*Var*⟩, ⟨*Const*⟩, and ⟨*Indiv*⟩ in `LParser.hs`. Also, the QuickCheck properties `prop1` and `prop2` live here.

Problems

❖ ***Problem 1 (100 points)*** ❖

The QuickTest test `prop1` tests your parser on fully parenthesized expressions. If your parser works on this, you get 100%.

✪ ***Challenge Problem 1: (10 points).*** ✪

The QuickTest test `prop2` tests your parser on expressions printed assuming the associativity and precedence rules given above. If your parser works on this, you get an additional 10%.

✪ ***Challenge Problem 2: (0 points, just glory).*** ✪

Extend the language to include another binary operator => (implies) that has precedent between + and &. Include convincing tests that your new parser works.

References

B. O'Sullivan, J. Goerzen, and D. Stewart. *Real World Haskell*. O'Reilly, 2008. URL http://book.realworldhaskell.org.