Parser MP

CS 443 — Spring 2015

Revision 1.0

Assigned

Due

Objectives and Background

In this machine problem you will write the parser for the TIGER language. We have studied LL and LR parsers, and even parser combinators.

For our parser, we are going to use an off-the-shelf parser library called `instaparse`. This library allows you to specify your grammar in BNF, and outputs a tagged vector notation.

1 The Code

To get you started, we have provided a simple parser that handles variable declarations. Here is the code, with some comments.

We begin with the namespace declaration. The `instaparser` library will use prefix `inst`, and the EDN library will use the prefix `edn`. We mainly want (edn/read-string) because it will use a safer version of CLOJURE's reader to convert strings to native types.

```
(ns tiger.parse
  (:require [clojure.edn :as edn]
            [instaparse.core :as inst]))
```

The parser takes a multi-line string with the grammar. Here are some of the conventions they use: an item in curly braces means zero or more instances. Something in single quote matches an exact word. We use this for keywords. If we surround something with angle brackets, the parser will "hide" the token, not returning it as part of the parse tree. We can also use CLOJURE's regular expression form (e.g, `#'[0-9]+'` matches integers) to handle lexical analysis for us.

Full documentation is available on the github page.

```
(def parser
  (inst/parser
   "S = {dec}

    dec = vardec
        | tvardec

    vardec = <'var'> <ws> id <':='> <ws> exp
    tvardec = <'var'> <ws> id <':'> <ws> id <':='> <ws> exp

    exp = int
    int = #'[0-9]+' <ws*>


    id = #'[a-z][a-z0-9]*' <ws>

    ws = #'\\s*'  ")
  )
```

The tree we get back may have a lot of extra things we don't want, so we can define a transformation function to clean things up. The following function rewrites `:int` nodes to have the string argument returned as an actual integer, removes the `:exp` tags, and removes the `:id` tags from within `:tvardec` and `:vardec` nodes.

```
(defn xformer [t]
  (inst/transform {:exp (fn [e] e)
                   :int (fn [i] [:int (edn/read-string i)])
```

```
                    :vardec (fn [[:id x] e] [:vardec x e])
                    :tvardec (fn [[:id x] [:id t] e] [:tvardec x t e])
                    } t))
```

Here's an example at the repl to illustrate what's happening.

```
tiger.parse> (parser "var x := 113")
[:S [:dec [:vardec [:id "x"] [:exp [:int "113"]]]]]
tiger.parse> (xformer (parser "var x := 113"))
[:S [:dec [:vardec "x" [:int 113]]]]
```

To put it all together, we compose the parser and the transformer.

```
(defn parse-tiger
  [input]
  (-> input
      parser
      xformer))
```

2 Your Work

The syntax for the TIGER language is given in appendix A of the text. Your work is to finish the parser.

I have given you the Midje tests I used to check the parser. There are 10 fact sets, and all but the first are commented out. Use them as documentation for what the parser should output, and uncomment them as you work on each stage of the parser. Here are the stages I used:

- Variable declarations with integers
- Function declarations with integers
- Type declarations, arrays, and records
- Variable expressions (with records and subscripts)
- Expressions: l-value, nil, parens, sequencing, no-value, string literal
- Expressions: negation, function call, arithmetic, comparison
- Expressions: Boolean operators, precedence tests
- Expressions: record, arrays, assignment
- Expressions: if-then-else, if-then, while, for, break
- Expressions: let