Machine Problem: Monadic Parsing

Overview

In this assignment you will use the monadic parsing primitives we built together in class to implement a parser for a simple subset of the C language.

Starter code repository

Claim your repository via the invitation link on the class homepage and `git clone` your starter code.

Next, before continuing, remember to sign the honor pledge in the "package.yaml" file so we know who your submission belongs to.

Details

The subset of C you will be parsing consists of properly formatted (but not necessarily semantically correct!) function definitions of the following form:

```
type func_name(type param, type param, ...) {
    type local_var, local_var, ...;
    type local_var, local_var, ...;
    ...
    var = VAR_OR_VALUE;
    var = VAR_OR_VALUE;
    ...
    return VAR_OR_VALUE;
}
```

Permitted `type`s are `int` and `char`, permitted strings for `param`, `local_var`, and `var` are identifiers which start with a lowercase character and are followed by alphanumeric characters, and `VAR_OR_VALUE` can either be an identifier as previously described or an integer.

The following are examples of valid input for your parser:

```
char foo1() { }

int foo2(char param1) { }

char foo3(char param1) {
  return param1;
}

char foo4(char param1) {
  return -1;
}

char foo5(char p1, char p2, int p3) {
  return 0;
}

char foo6(char p1, char p2, int p3) {
  char local1;
  return local1;
}

char foo7(char p1, char p2, int p3) {
  char l1, l2, l3;
  int l4, l5, l6;
  return -1;
}

char foo8(char p1, char p2, int p3) {
  char buf1;
  int n, m;
  char buf2;
  n = m;
  buf1 = 10;
  buf2 = -20;
  return 100;
}
```

Our language subset requires that all local variable declarations occur before any assignments, and that the return statement appears at most once, and only as the last statement in the function. Whitespace is not important, as in C.

Implementation details

All your code for this assignment should go into "src/MP4.hs".

You will implement the parser `funcDef :: Parser (String,[String],[String],String)`, which, when used to parse a string containing a valid function definition, will return a tuple containing:

1. The name of the function
2. A list of the names of the parameters of the function (if any)
3. A list of the names of the local variables declared within the function (if any)
4. The variable name or integer value returned by the function (as a string), or the empty string, if there is no return statement.

Invoking `parse funcDef` on the examples above, for instance, will return:

```
(("foo1",[],[],""),"")

(("foo2",["param1"],[],""),"")

(("foo3",["param1"],[],"param1"),""]

(("foo4",["param1"],[],"-1"),"")

(("foo5",["p1","p2","p3"],[],"0"),"")
```

```
(("foo6",["p1","p2","p3"],["local1"],"local1"),"")

(("foo7",["p1","p2","p3"],["l1","l2","l3","l4","l5","l6"],"-1"),"")

(("foo8",["p1","p2","p3"],["buf1","n","m","buf2"],"100"),"")
```

For invalid functions, the parser will fail with `Nothing`. Examples of these can be found in the test suite.

We highly recommend that you implement parsers for different parts of a function definition, which you can then sequence (inside a `do` block) to build your function parser. Here are some of the parsers you might want to consider building:

```
typeName :: Parser String -- a parser that recognizes "int" or "char"
paramList :: Parser [String] -- a parser for a parameter list
assignment :: Parser () -- a parser for a single assignment statement
varDecls :: Parser [String] -- a parser for variable declarations of a given type
```

Test Cases

"test/MP4Spec.hs" contains 17 separate test cases, each of which is worth 3 points. You can run the tests with the command `stack test`.

Submission

First, make sure you correctly signed and committed the "package.yaml" file. We won't be able to easily map your repository to your name if you don't!

To submit your work, simply commit all changes and push to your GitHub repository.

Last updated: Fri Apr 23 21:42:58 2021