Homework 5: Lectures 9 & 10

CS 440: Programming Languages and Translators, Spring 2020

Due Fri Feb 21, 11:59 pm

What to submit

Submit the Haskell program in a `*.hs` file. You can put the answers to the written problems in a separate file or as comments in the `*.hs` file (your choice). Please name the files something like `Doe_John_440_hw5.hs` or `pdf`, and if you have multiple files, put them into a folder named something like `Doe_John_440_hw5`, zip the folder, and submit the zip file.

Remember the new requirements: If you work alone, please say so in your submission. If you work in a group but aren't the person submitting the solution, then create a short file with the names of everyone in your group (including yourself), and submit that to Blackboard (in the homework 5 folder). These new requirements will make it easier for us to detect if someone forgot to put names down on the submission or didn't do the homework.

Problems [50 pts]

A. Lecture 9: (Scanners), Grammars, and Productions

Programming Assignment (Lecture 9) [24 pts]

For Homework 4, you extended a `capture` routine that does a regular expression match and returns the found string. E.g., assuming `identifier`, `operator`, `constant`, and `punctuation` are appropriate regular expressions, we can identify four parts to the input `xyz+17$`:

- `capture identifier "xyz+17$" = Just("xyz", "+17$")`, then
- `capture operator "+17$" = Just("+","17$")`, then
- `capture constant "17$" = Just("17","$")`, and finally
- `capture punctuation "$" = Just("$","")`

For this assignment, the goal is to write a scanner that repeatedly calls `capture` to find strings like the ones above and then calls constructor functions for a new `Token` datatype (these are not the same kind of tokens used in the `capture` code).

```
data Token = Const Int | Id String |
             Op String | Punct String | Spaces String deriving (...)
```

Calling `scan` should return a list of the result tokens, e.g., `scan "xyz+17$" = [Id "xyz", Op "+", Const 17, Punct "$"]`.

What to do:

- You are to take a skeleton of the solution, `HW_05_skel_440.hs` (attached to this pdf), and extend it as necessary to get `scan` to work. Parts of the program are complete: the `Token` definition, the top-level call of `scan` (it calls a helper `scan'`), some of the regular expressions to look for, and some of the `capture'`

routine that `capture` uses. You'll have to complete `scan`', add in the missing regular expressions, and complete `capture'` as necessary.

- `scan' rexps revtokens input` takes the regular expression at the head of `rexps` and runs `capture` with it on the input. If `capture` succeeds, then `scan'` should run a tokenizer function to turn the found string into a `Token`, and then it adds the token to the head of the (reversed) list of tokens found so far. (Constants and spaces are a bit complicated; see more discussion below). If `capture` returns nothing, you can skip the rest of the input and just return the tokens you've found.
- The list of regular expressions to try is called `scan_rexps`; it cycles through constants, identifiers, operators, punctuation, and spaces. (It's an infinite list so you'll run out of input before running out of regular expressions to try.) You need to define the missing regular expression functions.
- The `scan_rexps` list is actually a list of pairs: a regular expression and a tokenizer function: `String` → `Token`. Most tokenizer functions are just the datatype constructors (`Id`, `Op`, etc).,
- The important tokenizer function you have to write is for constants: You can't just call `Const` on a string like `"17"` , you have to `read` the string to get the `Int 17` that you can pass to `Const`.
- The other important point is that the final list of tokens returned by scan should not contain any `Spaces` tokens. You can do this in `scan'` by not adding a `Spaces` token to the result being calculated. Or you could add it in here but remove it when it's time to return the whole list of tokens to scan. (Or wherever else -- your choice.) During scanning, spaces delimit tokens: E.g., `scan "12 34" = [Const 12, Const 34]`.

Grading guide: [24 pts total]

- Complete tokenizer for `Const`: 3 pts
- Build tokens: 3 pts
- Add tokens to results: 3 pts
- Ensure Spaces tokens aren't in the result: 3 pts
- Fill in missing regular expressions: 4 points
- Capturing of regular expressions (const, add, or, empty): 4 pts altogether; in_set: 2 pts, star: 2 pts,
- Your submitted `*.hs` file should include the skeleton code plus all your new code. For testing, we should be able to start up ghci and copy/paste your `*.hs` file into it and run `scan`.

B. Lecture 10: Top-Down Parsing; Recursive Descent Parsing pt 1 [26 pts]

1. [6 pts] Take the `Lec_10_recognizer.hs` program and add a new rule for function calls, as factors. So we want calls like `x(x,x)`, `x(x*(x+x))`, and so on. Include `x()` as a legal call. There is one detail: To keep the parser predictive, we can't have $F \rightarrow$ `x` | `x` other stuff because then an identifier `x` won't tell us which disjunct to use. A similar problem came up with $E \rightarrow T \mid T + E$; you can solve it using the same idea (one clause with an option).

You just have to write out the rule; you don't have to implement it in the Haskell program. Grading: 2 pts each: Handling `x()`, `x(`*one expression*`)`, `x(`*many expressions separated by commas*`)`. Hint: Use a nonterminal for lists of expressions.

2. [20 pts] Below is a very ambiguous grammar for regular expressions using concatenation, disjunction, Kleene star, sets […], constants, empty, and parentheses. Terminal `c` stands for any constant (a symbol a-z, 0-9, etc.) but not the special symbols or-bar, star, brackets, ε, or parentheses. E.g., $R \rightarrow^*$ `c (c* [cccc])*`. Below, backslashes before a symbol mean that symbol literally, as a terminal symbol. (Typical regular expression handlers use this syntax.)

   $R \rightarrow R\ R \mid R\ \backslash\vert \mid R \mid R\ \backslash* \mid \backslash[\ Cs\ \backslash] \mid \backslash\varepsilon \mid \backslash(\ R\ \backslash)$

   $Cs \rightarrow$ `c` *Ctail*

   Ctail $\rightarrow$ `c` *Ctail*

   For this problem, rewrite the grammar so that it's LL(1) – i.e., we could write a recursive descent parser using the grammar. Grading: 4 pts each for the concatenation, or-bar, Kleene star, ε, and parenthesized expression rules.

