Modules; Recursive Descent Parsing, pt 2

CS 440: Programming Languages and Translators, Fall 2019

A. Modules

- A Haskell module is a container for functions, types, and typeclasses. A Haskell program is a collection of modules that includes a main program. Modules can import other modules in and can export items out.
- In a program you access a module using an `import` statement. `Data.Char` and `Data.List` are standard Haskell libraries:

```
import Data.Char             -- a library module for strings & chars
import Data.List             -- a library module for lists
import This.That.TheOther    -- a personal module
```

- Loading a module lets you use its items without qualifying the name. Below, the prompt is initially `Prelude>` (the standard initial library); loading `Data.Char` changes the prompt. to say we've loaded `Data.Char`. If we unload `Data.Char`, the prompt changes back.

```
Prelude> Data.Char.isLetter 'a'
True
Prelude> isLetter 'a'
   [error message about isLetter out of scope]
Prelude> :m Data.Char
Prelude Data.Char> isLetter 'a'
True
Prelude Data.Char> :m - Data.Char
Prelude> isLetter 'a'
   [error message about isLetter out of scope]
```

Library module Data.Char

- The `Data.Char` module contains a lot of functions, all centered on character operations. Some useful tests it contains includes `isLetter`, `isNumber`, `isAlpha`, `isAlphaNum`, `isSpace`, `isUpper`, `isLower` (all take a character and tell you if it's of some given category). There are also functions on characters, such as `toLower`, `toUpper.`

Library module Data.List

- Module `Data.List` supports various lists operations. We've seen some (e.g., `take`, `drop`). Some other helpful routines are are `takeWhile`, `dropWhile`, `span`. All three take test functions and a list of values. `takeWhile` returns the initial segment of values that pass the test, `dropWhile` returns the suffix of values found by `takeWhile`, and `scan` returns two sublists: the values that pass the test and the values that fail the test:
    - `takeWhile isLetter "abCD135jk?!@z" = "abCD"`
    - `dropWhile isLetter "abCD135jk?!@z" = "135jk?!@z"`
    - `span isLetter "abCD135jk?!@z" = ("abCD","135jk?!@z")`

Using our own modules

- To make a file something you can load, add a module declaration to its beginning

```
module ModuleName where
[... rest of file ...]
```

- The module name should be capitalized and should match the name of the file. E.g., module `Parse` would be in file `Parse.hs`. In another program, you can use `import ModuleName`; in `ghci` you can use `import ModuleName` or `:l ModuleName` to load the file.
- If the file is inside a folder relative to the directory you start ghci in, then add it to the module name (e.g., `Folder.Folder.ModuleName`). Do this everywhere (the module declaration, any import declaration, an explicit module name usage)

Example 1: *.hs files are in the folder you run ghci in

- In file `Capture.hs:`

```
module Capture where
-- etc.

capture rexp input = -- declare capture function
```

- In file `Capture_Tests.hs:`

```
module Capture_Tests where
import Capture

re = RE_or [ etc ]          -- for RE_or declared in Capture
re2 = Capture.RE_or [ etc ] -- explicitly asks for Capture's RE_or
```

- With `ghci`:

```
% ghci
Prelude> :l Capture_Tests
[ message about loading Capture ]
[ message about loading Capture_Tests ]
Capture_Tests> capture re "abc"
Just ("ab","c")
```

Example 2: *.hs files are below the folder you run ghci in

- Say we put the files in folder `Ltest`; then we add need to add `Ltest` as a qualifier everywhere we want to use the modules in it.

```
In file Ltest/Capture.hs:
module Ltest.Capture where
-- etc.

capture rexp input = -- same as before
```


- In file `Ltest/Capture_Tests.hs`:

```
module Ltest.Capture_Tests where
import Ltest.Capture

re = RE_or [ etc ]          -- for RE_or declared in Capture
re2 = Ltest.Capture.RE_or [ etc ]   -- explicit module name
```

- With `ghci`:

```
% ghci
Prelude> :l Ltest/Capture_Tests
[ message about loading Ltest.Capture ]
[ message about loading Ltest.Capture_Tests ]
Capture_Tests> t01 <-- make analogous to earlier example [2/18]
True
```

B. Recursive Descent Parsing (review)

- **Recursive descent parsing** uses mutually recursive routines, one for each grammar nonterminal in the language. Each nonterminal's parser tries to parse an instance of the nonterminal (and remove it from the prefix of the input symbols).
- Recursive descent parsing works on **LL(1)** languages, where the parse is deterministic (no backtracking) and reads the input left-to-right (first L) producing a leftmost derivation (second L) using **top-down parsing** (from the start symbol toward the final terminal string yielded).
- The 1 in LL(1) means we get one symbol of **lookahead** as we parse, the head of the list of symbols that make up the current input to parse. This makes the parser **predictive**.
- Traditional recursive descent parsing has problems with grammars that have **left-recursive** rules (rules where *Nonterminal* → same *Nonterminal* plus other stuff).
- Our example language was a traditional cut-down version of expressions:
  - $E \rightarrow T\ \ Ttail$
  - $Ttail \rightarrow +\ T\ Ttail \mid \varepsilon$ -- I.e., $E \rightarrow T\ \{+\ T\}^*$ or $E \rightarrow T\ (+\ E)^*$
  - $T \rightarrow F\ Ftail$
  - $Ftail \rightarrow *\ F\ Ftail \mid \varepsilon$ -- I.e., $T \rightarrow F\ \{*\ F\}^*$ or $T \rightarrow F\ (*\ T)^*$
  - $F \rightarrow \mathtt{x} \mid (\ E\ )$
- And in the previous lecture we had a recursive descent parser that simply recognized expressions. It returned a `Maybe [Symbol]` result: `Nothing` if the parse failed and `Just` *leftover_input* if the parse found an instance of the nonterminal and removed its symbols as the prefix of the input (and returned `Just` the suffix).
- In the recognizer:

```
-- Rule E -> T Ttail
--
parse_E :: Recognizer
```

```
parse_E input =
    case parse_T input of
        Nothing -> Nothing
        Just leftover -> parse_Ttail leftover
```

Building and Returning Parse Trees

- Let's now look at not returning a parse tree when we find a derivation of the input.
- We'll need a new datatype for parse trees, and a successful parse will return a pair with the parse tree and leftover input.

```
-- The parse tree structure follows the grammar structure
--
data Ptree =
    Empty                           -- The empty parse tree
    | Id String                     -- Identifiers
    | Exp Ptree Ptree               -- For Term/Term_tail
    | Term Ptree Ptree              -- For Factor/Factor_Tail
    | Ttail Symbol Ptree Ptree      -- for Symbol Term Ttail
    | Ftail Symbol Ptree Ptree      -- Symbol Factor Ftail
    | Factor Ptree                  -- for Factor id
    | Paren Ptree                   -- Factor (Expr)
    deriving (Eq, Show, Read)

-- As before, symbols are just characters
--
type Symbol = Char
type Input  = [Symbol]

-- Instead of Recognizer, we have a Parser.  Most of the parsers
-- are Parser Ptree -- they return a Ptree and leftover input.
--
type Parser t = Input -> Maybe (t, Input)
```

- For an expression, we look for a term and a term tail; if the term tail is empty, then we use an Empty parse tree for it in the parse tree for the expression.

```
-- For an expression, we look for a term and term_tail.
--
-- Grammar rule: E -> T  Ttail
--
parse_E :: Parser Ptree
parse_E input =
    case parse_T input of
        Nothing -> Nothing
        Just (term, input1) ->
            case parse_Ttail input1 of
                Nothing -> Nothing
                Just (ttail, input2) ->
                    Just (Exp term ttail, input2)
```

```
-- A term is a factor and factor tail.
--
-- Grammar rule: T -> F Ftail
--
parse_T :: Parser Ptree
parse_T input =
    case parse_F input of
        Nothing -> Nothing
        Just (factor, input1) ->
            case parse_Ftail input1 of
                Nothing -> Nothing
                Just (ftail, input2) ->
                    Just (Term factor ftail, input2)

-- A Ttail is either '+' with a term and term_tail or empty
--
-- Grammar rule: Ttail -> + T Ttail | ε
--
parse_Ttail :: Parser Ptree
parse_Ttail input =
    case next_symbol '+' input of
        Nothing -> parse_Empty input
        Just (symbol, input1) ->
            case parse_T input1 of
                Nothing -> Nothing
                Just (term, input2) ->
                    case parse_Ttail input2 of
                        Nothing -> Nothing
                        Just (ttail, left3) ->
                            Just (Ttail symbol term ttail, left3)
```

- In the `parse_E` routine above, notice that we check for the term tail being empty; if that's the case, then we return the term parse tree as is (we don't build an expression tree with the term and the empty parse tree). This is `Just` a convenience to make the parse trees a bit less complicated.

- I'm omitting the code for terms as factors followed by factor tails because it's almost exactly like the code for expressions as terms and term tails.
- A factor is an identifier or a parenthesized expression: $F \rightarrow$ `x` | `\(` E `\)`. To keep the case expressions from nesting too deeply, the parenthesized expression has been split off into its own routine

```
-- A factor is an identifier or parenthesized expression
--
-- Grammar rule: F -> x | paren_E
```

```
    --
    parse_F :: Parser Ptree
    parse_F input =
        case parse_id input of
            Just (idtree, input1) -> Just(Factor idtree, input1) [2/18]
            Nothing ->
                case parse_paren_E input of
                    Nothing -> Nothing
                    Just (paren_tree, input') ->
                        Just(Factor paren_tree, input')


-- Parenthesized expression
--
-- Grammar rule: paren_E -> ( E )
--
parse_paren_E input =
    case next_symbol '(' input of
      Nothing -> Nothing
      Just (_, input1) ->
          case parse_E input1 of
              Nothing -> Nothing
              Just (expr, input2) ->
                  case next_symbol ')' input2 of
                      Nothing -> Nothing
                      Just (_, left3) ->
                          Just (Paren expr, left3)
```

- This parser code is contained in `Lec_11_Parse_Tall.hs`

Shortening the Parse Trees

- Because of the rules $E \to T$, $T \to F$, and $F \to$ `id`, running `parse_E "x"` gives

  `Just (Exp (Term (Factor (Id "x")) Empty) Empty,"")`
- Similarly, `parse_E "x+y"` gives

  `Just (Exp (Term (Factor (Id "x")) Empty) (Ttail '+' (Term (Factor (Id "y")) Empty) Empty),"")`
- We can shorten the height of parse trees (and make them more readable) by simply returning `Just(Id "x","")` for `parse_E "x"`. Similarly, `parse_E "x+y"` could return

  `Just (Exp (Id "x") (Ttail '+' (Id "y") Empty),"")`
- To do this, one way is to modify `parse_E` and `parse_T` to look for and avoid these situations.
- `parse_E` still needs to run parse_Ttail input1 (and yield Nothing if term tail yields Nothing), but it can check for a `ttail` of Empty, which indicates that we're in efect, we're using the rule $E \to T$, not $E \to T\ Ttail$. In that case, we can return `Just(term, input1)` .
- The most straightforward way to do this is to find the case match for `Just (ttail, input2) ->` … and insert another case before it to check for `Empty`:

```
case parse_Ttail input of
    Nothing -> Nothing
    Just (Empty, input2) -> Just(term, input1)
    Just(ttail, input2) -> … as currently …
```

- A similar change goes into `parse_T`: Insert the case `Just(Empty,input2) -> Just(factor, input1)` before the general case `Just(ftail, input2) -> …`
- (Making these changes is an activity question; see below)
- So this way checked for an empty tail before making the larger expression or term expression.
- An alternative technique is to change the function used to make expressions and terms.
  - When parsing an expression, instead of calling `Just (Exp term ttail, input2)`, call `Just (make_tail Exp term ttail, input2)`, where `make_tail` looks at the term tail argument; if it's `Empty`, then `make_tail` just returns the `term` argument. Otherwise, `make_term` calls the `Exp` constructor on the term and term tail, as currently.
  - Similarly, when parsing a term, use `Just (make_tail Term factor ftail, input2)`, where if the ftail argument is `Empty`, then make_tail returns the `factor` argument. Otherwise, make_term calls the `Term` constructor on the factor and factor tail.
- (Making this set of changes is also an activity question; see below.)

Activity Questions for Lecture 11

Changes to Parse_Short_activity.hs

1. Go into `parse_E` and `parse_T` and make the parse trees shorter if the term or factor tail is `Empty`. In `parse_E`, add a `case` to the `parse_Ttail input1` check: if we get `Just (Empty, input2)` then simply return `Just (term, input1)`. Similarly, in `parse_T`, after calling `parse_Ftail`, use a new case to see if the factor tail matches `Empty`; in that case return `Just (factor, input1)`. Run parse_T "x" and parse_E "x" to verify that your changes work.

2. Repeat the previous problem with a different change. (First go back and comment out the changes for Problem 1.) Take `Just (Exp term …)` and `Just (Term factor …)` and insert a call to `make_tail` to get `Just (make_tail Exp term …)` and `Just (make_tail Term factor … )`. Then find the stub for `make_tail` and complete it: given a call `make_tail builder ptree1 ptree2`, if ptree2 matches Empty, simply return `ptree1`; otherwise return the result of calling the builder function on `ptree1` and `ptree2`. Once again, try running `parse_T "x"` and `parse_E "x"` to verify your solution. Include the type of `make_tail` in your new code.

3. Why would it be a mistake to to go into `parse_Ttail` and try to shorten its parse trees by replacing
```
Just (Ttail symbol term ttail, left3)
```
with
```
Just (make_tail (Ttail symbol) term ttail, left3)    ?
```

4. Why do we need parentheses around `Ttail symbol` in the previous question?

5. Add a new kind of parse tree `data Ptree = … | Negative Ptree` and modify the grammar for Factor:

    Factor → `id` | – *Factor* | (*E*)

    If a minus sign appears, then build and return the `Negative` of the factor parse tree.

Solution to Selected Activity Problems

1. (Check for `Empty` tail in parse_E and parse_T to shorten the parse trees)

```
parse_E input =
    case parse_T input of
        Nothing -> Nothing
        Just (term, input1) ->
            case parse_Ttail input1 of
                Nothing -> Nothing
                Just (Empty, input2) -> Just (term, input1)
                Just (ttail, input2) -> Just (Exp term ttail, input2)

parse_T input =
    case parse_F input of
        Nothing -> Nothing
        Just (factor, input1) ->
            case parse_Ftail input1 of
                Nothing -> Nothing
                Just (Empty, input2) -> Just (factor, input1)
                Just (ftail, input2) -> Just (Term factor ftail, input2)
```

2. (Use `make_tail` in `parse_E` and `parse_T` to shorten the parse trees)

```
parse_E input =
    case parse_T input of
        Nothing -> Nothing
        Just (term, input1) ->
            case parse_Ttail input1 of
                Nothing -> Nothing
                Just (ttail, input2) ->
                    Just (make_tail Exp term ttail, input2)

parse_T input =
    case parse_F input of
        Nothing -> Nothing
        Just (factor, input1) ->
            case parse_Ftail input1 of
                Nothing -> Nothing
                Just (ftail, input2) ->
                    Just (make_tail Term factor ftail, input2)

make_tail :: (Ptree -> Ptree -> Ptree) -> Ptree -> Ptree -> Ptree
make_tail _ ptree1 Empty = ptree1
make_tail builder ptree1 ptree2 = builder ptree1 ptree2
```


3. (Add make_tail to `parse_Ttail`?)

   The problem with

   `Just (make_tail (Ttail symbol) term ttail, left3)`

   is that if `ttail` is `Empty`, then we'd return `Just(term, left3)`, which means we'd omit the plus symbol before the term. E.g., instead of parsing "+x" as (`Ttail '+' (Id "x") Empty`), we'd parse it as `Id "x"`. (A similar problem would occur if we added make_tail to `parse_Ftail`.)

4. Without the parentheses around `Ttail symbol`, the `make_tail` call would have four arguments, which is a type error.

5. (Negative factors) This is what you get if parse $F \rightarrow id \mid$ `-`$F \mid$ `\(` E `\)`

```
parse_F input =
    parse_id input `fails`( \() ->
    next_symbol '-' input `bind` (\(minus, input1) ->
    parse_F input1          `bind` (\(factor, input2) ->
    Just (Negative factor, input2) ))
                            `fails` (\() ->
    parse_paren_E input ))
```

   Reordering the rules as $F \rightarrow id \mid$ `\(` E `\)` $\mid$ `-`F makes for code that's a little easier to read (my opinion):

```
parse_F input =
    parse_id input          `fails` (\() ->
    parse_paren_E input     `fails` (\() ->
    next_symbol '-' input   `bind` (\(minus, input1) ->
    parse_F input1          `bind` (\(factor, input2) ->
    Just (Negative factor, input2) ))))
```

1. (Fill out `Parse_T` and `parse_Ftail`) These routines are analogous to `parse_E` and `parse_Ttail`

```
parse_T input =
    parse_F input       `bind`  (\ (factor, input1) ->
    parse_Ftail input1  `bind`  (\ (ftail, input2) ->
    Just (make_tail Term factor ftail, input2) ))

parse_Ftail input =
    next_symbol '*' input       `bind` (\ (symbol, input1) ->
    parse_F input1              `bind` (\ (factor, input2) ->
    parse_Ftail input2          `bind` (\ (ftail, left3) ->
    Just (Ftail symbol factor ftail, left3) )))
                                `fails` (\() ->
    parse_Empty input )
```

2. If we take out the `make_tail` in `parse_T`, then the `Tail factor ftail` that remains builds a taller parse tree (if `ftail` is empty, we get `Term factor Empty`).

3 (Rewrite `parse_id` using `bind` instead of `case`)

```
parse_id input =
    getId (dropSpaces input) `bind` (\(idstring, input1) ->
    Just(Id idstring, input1) )
```

4. (Results of `bind` (`Just` *x*) `Just`)

For all values, `bind` (`Just` *x*) `Just` = `Just` *x*. From the `bind` definition `bind` (`Just val`) `f` = `f val`, we get (by referential transparency) that `bind` (`Just` *x*) `Just` = `Just` *x*.