Modules; Recursive Descent Parsing, pt 3

CS 440: Programming Languages and Translators, Spring 2020

A. Concentrating on the non-Nothing cases in the Recursive Descent Parser

- The parser code above gets pretty deeply nested with `case` expressions, almost all of which are of the form (`case` *exp* `of Nothing` -> `Nothing` ; `Just` (`tree`, `leftover`) -> *code using* `tree`, `leftover`)
- We can make the code simpler by only writing the useful code (takes a `tree` and `leftover` and does something with them).
- The bind routine below takes a function and a `Maybe` value; if the value is `Nothing`, it returns `Nothing`; if the value is `Just` *val*, it runs the function on *val*.

```
bind :: Maybe a -> (a -> Maybe b) -> Maybe b
bind Nothing f = Nothing
bind (Just val) f = f val
```

- Using bind, instead of writing

```
case expr of
    Nothing -> Nothing
    Just val -> … computation involving val …
```

we can use

```
bind expr fcn
```

where the "**continuation**" function is defined as

```
fcn val = … computation involving val …
```

A first attempt

- As a first try at simplifying our code, here's parse_paren defined using bind and some assistant functions:

```
-- *** Doesn't work **
parse_paren_E input        = bind (next_symbol '(' input) level1
level1 (lparen, input1)    = bind (parse_E input1) level2
level2 (expr_tree, input2) = bind (next_symbol ')' input2) level3
level3 (rparen, input3)    = Just (expr_tree, input3) -- ** ERR **
```

---------- 2020-02-18 [roughly, before the rewrite]

- The above doesn't work because `level3` wants to use `expr_tree`, which is part of `level2`, not `level3`.
- We'll look at different ways to solve this problem and work our way up to a reasonably simple solution.

A second attempt

- If we embed the definition of `level3` within `level2`, then `level3` can use `expr_tree`. Since level2 doesn't use any nonlocal variables, we don't need to embed it within `level1`. (Note I've given it a different name: `parse_paren_E2`.

```
parse_paren_E2 input =
    let level1 (lparen, input1) =
```



```
            bind (parse_E input1) level2
        level2 (expr_tree, input2) =
            let level3 (rparen, input3) = Just (expr_tree, input3)
            in bind (next_symbol ')' input2) level3
    in bind (next_symbol '(' input) level1
```

A third and fourth attempt

- For completeness, here each level is completely embedded in the one above it. (Wow, this is ugly ?!)

```
parse_paren_E3 input =
    let level1 (lparen, input1) =
          let level2 (expr_tree, input2) =
                let level3 (rparen, input3) =
                      Just (expr_tree, input3)
                in bind (next_symbol ')' input2) level3
          in bind (parse_E input1) level2
    in bind (next_symbol '(' input) level1
```

- The `level1`, `level2`, and `level3` functions only get called once each, so it's worth trying to replace them with unnamed lambdas. This version does that, using `bind` in prefix. Just for an alternate style, each closing right paren appears in the same column as its left paren (if it doesn't go on the same line as the left paren).

```
parse_paren_E4 input =
    bind (next_symbol '(' input)
        (\(lparen, input1) ->
            bind (parse_E input1)
                (\(expr_tree, input2) ->
                    bind (next_symbol ')' input2)
                        (\ (rparen, input3) ->
                                Just (expr_tree, input3)
                        )
                )
        )
```

The final solution

- The final solution I'll present uses the same code but formats it differently. The lambda headers (which name the parameters) are on the right; actions we want to carry out are on the left.

```
parse_paren_E5 input =
    (next_symbol '(' input)     `bind ` (\ (lparen, input1) ->
    (parse_E input1)            `bind ` (\ (expr_tree, input2) ->
    (next_symbol ')' input2)    `bind ` (\ (rparen, input3) ->
    Just (expr_tree, input3) )))
```

- The different versions of parse_paren are all contained in a file you can load into ghci. It's called `Using_bind.hs` (attached to this handout). It loads in the `Parse_Short` code automatically. Here's output from `ghci` showing how the five different versions of `parse_paren_E` all produce the same output.

```
Prelude> :l using_bind.hs
[1 of 2] Compiling Parse_Short      ( Parse_Short.hs, interpreted )
[2 of 2] Compiling Main             ( using_bind.hs, interpreted )
Ok, two modules loaded.
*Main> parse_paren_E "(x+y*z)"
Just (Exp (Id "x") (Ttail '+' (Term (Id "y") (Ftail '*' (Id "z") Empty)) Empty),"")
*Main> parse_paren_E2 "(x+y*z)"
Just (Exp (Id "x") (Ttail '+' (Term (Id "y") (Ftail '*' (Id "z") Empty)) Empty),"")
*Main> parse_paren_E3 "(x+y*z)"
Just (Exp (Id "x") (Ttail '+' (Term (Id "y") (Ftail '*' (Id "z") Empty)) Empty),"")
*Main> parse_paren_E4 "(x+y*z)"
Just (Exp (Id "x") (Ttail '+' (Term (Id "y") (Ftail '*' (Id "z") Empty)) Empty),"")
*Main> parse_paren_E5 "(x+y*z)"
Just (Exp (Id "x") (Ttail '+' (Term (Id "y") (Ftail '*' (Id "z") Empty)) Empty),"")
```

Studying the bind code; the Maybe Monad

- To review: The `bind` routine takes a `Maybe` value and a function that expects an actual value (not a `Maybe` value). If the `Maybe` value holds an actual value (`Just` *value*), then it calls the function on that value.
- Using `bind` here lets us take the code pattern `case` *expr* `of Nothing -> Nothing` and rewrite it just mentioning what happens if we pass in an actual value; The repetitive `Nothing -> Nothing` code is bundled up inside of `bind`.

  ```
  bind :: Maybe a -> (a -> Maybe b) -> Maybe b
  bind Nothing f = Nothing
  bind (Just val) f = f val
  ```

- Using Maybe in this way is an example of a more general pattern called a **monad**. We'll look at monads in more detail later in the semester, but briefly, there are two parts to a monad.
  - A way to modify or augment data. (For `Maybe`, it was by using `Just` or `Nothing`.)
  - A `bind` function of type (*modified data*) -> (*unmodified data* → *fcn result*) → *possibly fcn result*.
  - In general, `bind` has the job of taking modified data and trying to retrieve the original unmodified data from it. If successful, it calls the function to get a result. If `bind` is unable to retrieve unmodified data, then it has to do something else.
  - For `Maybe`, the `bind` routine looks for the unmodified data in a `Just` expression. The bind call returns tries to return the result of the function (called un unmodified data), but if there's no data (the `Nothing` alternative), bind returns something else (`Nothing`, in our case).
- All `bind` routines just apply the function to actual data (if bind can find that data). Different monads modify data in different ways, so they require different `bind` routines to access data. They also need to return some sort of value if there was no actual data.

What if we don't want Nothing?

- Our `bind` routine always returns `Nothing` if it's given `Nothing`. This is fine if we're trying to sequence some actions. But what do we do if we want to make a choice between `Nothing` and `Just` *val*?

- This is the opposite of `bind`, which serves as a pipe for `Nothing` but calls a function if given `Just` *val*.
- We want a routine that pipes through `Just` *val* but calls a function if given `Nothing`. It's called `fails`:

```
fails :: Maybe a -> (() -> Maybe a) -> Maybe a
fails Nothing f = f()
fails ok _ = ok
```

- The function call `f()` looks strange: We're passing a zero-tuple to `f`. The zero-tuple, spelled () and sometimes pronounced "nil" is handy if you need a value for syntax's sake but don't actually need the value.
- The zero-tuple has a type that's also spelled `()`. The type is what's used in `(() -> Maybe a)`, the type of function that's the second argument to `fails`.
- As an example of using `fails`, let's look at the basic parse factor routine: It tries to parse an identifier, and if that fails, it tries to parse an parenthesized expression. Using `fails`, we can write it as:

```
parse_F3 input =
    parse_id input `fails` (\() ->
    parse_paren_E input )
```

- We run `parse_id` on the input; if that succeeds and produces `Just` a parse tree and leftover input, then the `fails` routine just yields that. But if `parse_id` returns `Nothing`, then `fails` calls the lambda function (using argument `()`), which calls `parse_paren_E` on `input`. (Note the body of the (\() -> …) function is part of the body of `parse_F3`, so it has access to the parameters of parse_F3.

Activity Questions for Lecture 12

Changes to Parse_Bind_Fail_activity.hs

1. Replace the stubs for `Parse_F` and `parse_Ftail` with working code..

2. What happens if you remove the `make_tail` call in your answer to question 1?

3. Rewrite `parse_id` using `bind` instead of `case`.

4. Try evaluating `bind` (`Just` *x*) `Just` for various values of *x*. Why does it do what i does?

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. (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*.

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

```
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 $F \to id \mid$ `\(` E `\)` $\mid -F$ 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) ))))
```