

CS51 Assignment 6:

λ Tunes

Due: Tue, April 7th, 2009, 11:59 PM

Total Points: 55 (including 5 style points)

In this assignment, you will use the parser combinators we learned about in lecture to parse an input file. You will begin by writing a Backus-Naur Form grammar describing the proper format for a specific type of input. Then, generalizing to arbitrary structures represented in flat XML text files, we will use the magic of parser combinators to bring these structures to life in our code.

You should refer to Lecture 11 and Section 6 notes for more information about parsing in Scheme. The file `parse.ss`, which is included at the top of the Scheme file, has all of the parser combinators and other functions you'll need for this assignment, including those used below. (You may not need all of the combinators defined in the file.) You can download the file on the website in order to see the definitions of the combinators, or look in the lecture notes.

Note: this assignment may seem extraordinarily long – don't panic! This is because we've written lots and lots of code for you, as well as duplicated much of the code from the .scm file in the PDF. You have to make sense of it, but only have to write a very small amount of code yourself.

1 Understanding Parsing

In Project 1b, we wrote a function `valid-parse?` that told us whether a sentence was valid in the pirate grammar. However, in Lectures 11 and 12, we learned that `valid-parse?` was not actually a parser – it was, as we now know, a recognizer, since it only returned `#t` or `#f` depending on whether an input was in the language. In this problem set, we will see why a parser can be more useful than a recognizer: when a parser processes valid input, rather than just returning `#t`, a parser can return a value that represents the structural meaning of the input.

For example, consider Google's search box as the input field for a parser. Everyone knows about the neat trick where you type a valid arithmetic expression, and Google solves it for you. In order to do this, Google must know that it should evaluate "1+1", but search for

“1++1”. A recognizer for valid arithmetic expressions can tell it that. But given that has a valid arithmetic expression, like “1+1” or “3+5*7” or “2*the answer to life, the universe, and everything”, how does it actually evaluate it? It needs to take the input string “3+5*7” and *parse* it, perhaps creating a rooted tree like:

```

      +
     / \
    3   *
     / \
    5   7

```

or, more concisely,¹

```
(+ 3 (* 5 7))
```

Given the input in this form, it’s fairly easy to write a function that evaluates it as an arithmetic expression. (Try it!)

Parsing is especially useful in compiling programming languages, where (as with arithmetic expressions) a parser takes source code and returns an *abstract syntax tree*: i.e., a tree like the above, but with arbitrary operators and language constructs at the nodes. In last year’s problem set, we wrote a parser for Scheme in C++. This year, we’re going to be...

2 XML Grammar

(6 points total)

... writing a parser for XML in Scheme! You thought we were going to say “writing a parser for C++ in Scheme”, didn’t you? Do we really seem that cruel?

XML is a way to organize hierarchical information in text files. XML has a nested, naturally tree-like structure, where information is stored between tags that identify it. We could go on describing XML for quite a while, but the best way to understand it is actually to see it in action. For example, here is some XML describing a song:

```

<song>
  <title>Boten Anna</title>
  <artist>Basshunter</artist>
</song>

```

¹Does the following syntax look familiar? Scheme lists – and, by extension, Scheme programs – are essentially trees of the above form. This explains (1) why Scheme is so easy to parse, (2) why there are so many annoying parentheses, and (3) why Scheme code is so good at manipulating Scheme code!

If we wanted to describe another song, I would repeat the entire structure and fill in different information inside the fields. Pretty easy, huh? (At least easier than C++!)

For an overview of XML, you may visit the Wikipedia page. You can look through the XML files that we've provided for an example of using XML to describe a music library. Be careful – we're only promising that the first one is properly formed; the others may (or may not) be examples of improperly formed input that your parser should detect and refuse to parse.

At the bottom of the code for this assignment, you've been given some sample playlists. Some are well-formed XML playlists and some are not, for a variety of reasons. You may use them as test cases; the only thing we'll tell you is that the first one is well-formed. (Although you should be able to manually determine whether or not they're valid; otherwise, they wouldn't be useful test cases!)

Exercise 1.

4 points

Now we'll give you an English description of the syntax for the specific subset of XML that we want to parse, and you'll write part of the BNF description for it. Remember some common pitfalls of writing grammars for recursive-descent parsers:

- Left recursion: when a nonterminal symbol occurs first in its own expansion, with no intervening terminal symbols matched in the string
- Ambiguity: when a string can be parsed two or more ways according to the grammar
- Search order: when a shorter portion of the input is matched than is actually intended; if the parser had kept reading using a different rule, it would have found a longer match.

Description:

- A **playlist** can have any number of **entry(s)**.
- An **entry** will have a **name**, a **filename**, and an **artist**. It may or may not also have an **album** and/or a **picture**. For the purposes of writing this grammar, you may assume these fields will always appear in this order – **name**, **filename**, **artist**, **album** (if present), **picture** (if present) (although our code below will accept them in any order).
- A **filename** ends in ".mp3". A **picture** ends in ".png" or ".jpg".
- A **filename** and a **picture** can contain both numeric and alphabetic characters, as well as dots (.), but no whitespace. All other fields can only contain alphabetic characters and whitespace.
- An XML field will open with a <field> tag and close with a </field> tag. Field names are nonempty sequences of alphabetic characters only.

- Arbitrary whitespace (newlines, tabs, spaces) may appear between tags and inside the bodies of fields, but none may appear within an opening or closing tag.

(a) [4 points]

Fill in the blanks to finish this BNF grammar. Remember the $*$ operator, denoting any number of the symbol indicated; the $+$ operator, denoting at least one of the symbol; and the $?$ operator, denoting either 0 or 1 of the symbol. (Note: $---$ below represents any number of things that you need to fill in, not necessarily just one.)

```

playlist ::= ws '<playlist>' entry* '</playlist>' ws
entry ::= ws '<entry>' ___
name ::= ___ '<name>' ___ '</___>'
artist ::= ___
filename ::= ws '<filename>' ((c | n)* '.')+ 'mp3' '</filename>'
album ::= ___
picture ::= ___
ws ::= w*
w ::= ' ' | '\t' | '\n'
c ::= 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
n ::= '0123456789'

```

Having a grammar like this is great if the structure of our files is simple, but it quickly becomes unwieldy if we want to extend it. For instance, suppose we wanted the `name`, `artist`, etc. fields to appear in an arbitrary order (not necessarily sequential): then we would have to write out an alternative for every possible ordering of these fields, in order to ensure that there is exactly one of each. A better solution is just to give up on enforcing *semantic* information – i.e., the meanings of the XML tags – in the parsing step. Instead, we can write a parser for XML² in general, and then write post-processing functions to check whether the result of the parse is valid in the specific language we want. This will be surprisingly easy since, as noted above, XML has a natural tree structure, which we will simply parse into tree-like Scheme expressions.

Exercise 2.

2 points

The following grammar *almost* represents XML:

```

xml ::= ws '<' id '>' xml-body '</' id '>' ws
xml-body ::= xml+ | (c | n | w | .)+ | ''
id ::= c+
ws ::= w*
w ::= ' ' | '\t' | '\n'

```

²Actually, a somewhat restricted subset of XML – but enough to get the flavor.

```
c ::= 'abcdefghijklmnopqrstuvwxyABCDEFGHIJKLMNOPQRSTUVWXYZ'
n ::= '0123456789'
```

(a) [2 points] What's the problem: why doesn't this grammar actually represent (only) valid XML?

In fact, no BNF grammar can represent precisely XML – in other words, XML is not a *context-free language*. This won't stop us from parsing it, however! We can take the above BNF as an informal description, write a parser based on that, and then do postprocessing to ensure that we're left with only the parses we want.

3 Parsing

(12 points total)

Our XML parser, like any parser combinator, should take a list of characters – the input being parsed – and return either a **fail** struct (if the input is not valid XML) or an **ans** struct (if the input is valid XML).

Recall from lecture the definitions of these two:

```
(define-struct ans (result unconsumed))
(define-struct fail ())
```

The **unconsumed** part of the **ans** is just the (possibly empty) list of characters remaining after we have removed the characters corresponding to a valid XML field. For the **result**, in this case, we'll want a struct that represents the actual structure of XML:

```
(define-struct xfield (name body)) ; An XML field.
```

where the **name** of a field is its tag's name, and the **body** of a field is the stuff contained between its opening and closing tags – this could be a string, or a list of zero or more **xfields**.

For example, the output of parsing:

```
<song>
  <title>Boten Anna</title>
  <artist>Basshunter</artist>
</song>
```

should be an **ans** struct whose **result** is:

```
(make-xfield "song"
  (list (make-xfield "title" "Boten Anna") (make-xfield "artist" "Basshunter")))
```

Now, John Scheme has already set out to write an XML parser. He feels quite confident – he took CS 50, after all, and moreover he just finished reading *A Gentle Introduction to Parser Combinators*. Below is John's code – his Scheme interpreter was suffering from an

acute case of symbolism when he was composing it, so you may need to connect a few dots in order to make it work.

Exercise 3.*12 points*

(a) [12 points] Uncomment the function definition below and fill in the missing combinators so that it parses XML correctly. Each underscore block represents exactly one parser combinator or other atom.

```
(define (xml-pain l)
  (let* ([r ((cat ___ (char #\<)) 1)]; read leading whitespace, <
        [r1 (if (fail? r) r ((plus alpha) (ans-unconsumed r)))] ; tag name
        (if (fail? r1) ___
            (let* ([name (list->string (ans-result r1))]
                  [r2 ((char #\>) (___ r1))]; read >
                  (if (fail? r2) r2
                      (let* (; parse body
                            [r3 ((plus ___) (ans-unconsumed r2))]
                            ; is the body one or more xml fields?
                            [r4 (if (fail? r3)
                                    ; if not, perhaps it's a nonempty field body
                                    ((___ list->string (plus (alt alpha white)))
                                     (ans-unconsumed r2)) r3)]
                            [r5 (if (fail? r4)
                                    ; if not, it must be empty (a list of zero xml fields)
                                    (always (ans-unconsumed r2)) r4)]
                            ; read </
                            [r6 (if (fail? r5) r5 ((str "</") (ans-unconsumed r5)))]
                            ; read tag name and >
                            [r7 (if (fail? r6) r6 ((cats (list (___ ___) (char #\>) ws))
                                     (ans-unconsumed r6)))]
                            (if (fail? r7) r7 (___ (make-xfield name (ans-result ___)
                                                         (ans-unconsumed r7))))))
                    ; return a field with given tag name and body, if successful
```

That was painful! It's a good thing we have parser combinators so that we don't have to work with code like that regularly.

4 Higher-Order Style*(16 points total)*

When making parsing code more elegant, it's often a good idea to think about writing fancier combinators.

Exercise 4.*5 points*

Here's an additional combinator, `cats-n` (included in `parse.ss`), that may help us in

rewriting our parser.

```
(define (cats-n n lst)
  (if (= n 0)
      (catl (car lst) (cats (cdr lst)))
      (catr (car lst) (cats-n (- n 1) (cdr lst)))))
```

(a) [5 points] Write a short description of the meaning of `cats-n` (at most three sentences). (Note: simply paraphrasing the combinator’s implementation isn’t valid.)

Now, here’s an attempt at writing our XML parser in elegant, higher-order style – unfortunately, it is not valid Scheme:³

```
(define xml
  (cats-n
   4
   (list ws
         (char #\<)
         (plus alpha) ; When invoked on the input char list, this will yield
                       ; an answer struct, whose result is a list of chars
                       ; representing the name of the tag we’re starting
                       ; to parse...
         (char #\>)
         (pmap (lambda (r) (make-xfield name r)) xml-body)
         ; Here we use the name of the tag, as we combine it with the
         ; result of parsing the rest of the tag body recursively...
         (str "</")
         (str name) ; We use it again here, in checking that the closing
                   ; tag matches the opening tag above...
         (char #\>)
         ws))))
(define xml-body
  (alts (list (plus xml)
              (pmap list->string
                    (plus (alts (list alpha digit white (char #\.)
                                  always))))))
```

The problem is that `name` is never defined – `(plus alpha)` returns an `ans` whose result is the name of the tag, which we promptly throw away, and can’t use in the resulting combinators! What we really want is for `(plus alpha)` to feed its output result, the name of the tag, to the rest of the combinators in `xml`. Here are the rest of the combinators (appropriately catted together to return what we want):

```
(cats-n
 1
 (list (char #\>)
       (pmap (lambda (r) (make-xfield name r)) xml-body)
```

³If you’re feeling really ambitious, a good exercise would be to skip over the following derivation and just try to write a working parser based on the combinators `cats-n` (above) and `pat-k`, `pat-k-str` (below).

```
(str "</")
(str name)
(char #\>)
ws))
```

As we can see, this is a perfectly fine combinator that we could `cat` onto the above, except that `name` is undefined. But if we assume that someone passes us `name`, we can write a *function* yielding the appropriate combinator:

```
(lambda (name)
  (cats-n
   1
   (list (char #\>)
         (pmap (lambda (r) (make-xfield name r)) xml-body)
         (str "</")
         (str name)
         (char #\>)
         ws)))
```

This is known as a *continuation*, because it specifies – given a parameter, in this case the name of the tag – the proper way to continue the expression (in this case, the resulting combinator). So, if we could just get `(plus alpha)` to feed its output to this continuation, we would be all set. In fact, we’ll do one better: we’ll write a combinator, `pat-k`, that takes *any* combinator `p` – such as `(plus alpha)` – and converts it to take a continuation, `k`, and pass the result of `p` along to that continuation.

```
(define (pat-k p)
  (lambda (k) ; (a continuation)
    (lambda (cs) ; We return a parser combinator, i.e. a function taking
      ; a list of characters cs...
      (let ([r (p cs)]) ; which first applies the pattern p to cs,
        (if (fail? r)
            r
            ((k (ans-result r)) ; then gives p's result to the
              ; continuation k, yielding
              ; another parser combinator...
              (ans-unconsumed r)))))) ; which we finally apply
      ; to the remainder of
      ; the string.
```

We’re almost there! There’s just one subtle bug in the following attempt:

```
(define xml
  (cats-n
   2
   (list ws
         (char #\<)
         ((pat-k (plus alpha))
          (lambda (name)
```

```

      (cats-n
        1
        (list (char #\>)
              (pmap (lambda (r) (make-xfield name r)) xml-body)
              (str "</")
              (str name)
              (char #\>)
              ws))))))
(define xml-body
  (alts (list (plus xml)
             (pmap list->string
                  (plus (alts (list alpha digit white (char #\.)
                                always)))))))

```

The problem is that our continuation is expecting the tag name, `name`, as a string – but it would be passed a list of characters, since that’s the result of a successful parse by `(plus alpha)`. We could just use `(string->list name)` instead of `name` in the body of the `lambda`, but in the spirit of greater generality, let’s extend `pat-k`:

```

(define (pat-k-str p)
  (lambda (k) ; (a continuation expecting a string)
    ((pat-k p) (lambda (cs) (k (list->string cs))))))

```

Now we can finally write our elegant XML parser:

```

(define xml
  (cats-n
    2
    (list ws
          (char #\<)
          ((pat-k-str (plus alpha))
           (lambda (name)
             (cats-n
              1
              (list (char #\>)
                    (pmap (lambda (r) (make-xfield name r)) xml-body)
                    (str "</")
                    (str name)
                    (char #\>)
                    ws)))))))))
(define xml-body
  (alts (list (plus xml)
             (pmap list->string
                  (plus (alts (list alpha digit white (char #\.)
                                always)))))))

```

Make sure you understand what this code is doing. (What’s the body of the function defined by `(lambda (name) ...)`? Why did we write it that way?) You may want to stare at it for a bit.

To make sure we've got the hang of these parser combinators, let's write parsers for a couple of languages that are a lot like XML, but subtly different.

Exercise 5.*11 points***(a)** [3 points]

Write a parser for a language like XML, except that tags are closed (Bash-script-style) by the reversal of their names. Thus, `<foo>bar</foo>` is not valid, but `<foo>bar</oof>` is valid. (For this part, you can just copy and paste the code for `xml` and `xml-body` above, and modify it slightly; if you're writing more than a few extra atoms/parentheses, stop and rethink what you're doing.)

(b) [8 points]

Write a parser for a language like XML, except that all tags must be the same as the outermost tag. Thus, `<foo> <bar>baz</bar> <bar>qux</bar> </foo>` is not valid, but `<foo> <foo>baz</foo> <foo>qux</foo> </foo>` is valid. A skeleton is filled in for you. With reasonable amounts of copy/pasting, you should not need to write very much code.

5 Back to Playlists*(15 points total)*

Now we have two different parsers for general XML. Great! However, we still want to restrict ourselves to things that represent valid playlists. Our parser would accept

```
(string->list "<gideon>says_hi</gideon>")
```

as a valid `entry`, which doesn't make semantic sense for our purposes, even though it's syntactically valid XML.

First, let's look at the structs that we want to be parsing our XML into. A `playlist` struct should be a list of `entry` structs. An `entry` struct should contain 5 fields: `name`, `file`, `artist`, `album`, and `picture`. We've also written `empty-playlist`, as well as the function `(add-entry ...)`, which adds a song entry to a playlist. Of course, if we are given something that is not valid XML for a playlist, we should return a `fail` struct.

```
(define-struct playlist (entries))

(define empty-playlist (make-playlist '()))

(define-struct entry (name filename artist album picture))

(define (playlist-add-entry e p)
  (make-playlist (cons e (playlist-entries p))))
```

Now, let's think about what kind of post-processing we need to do to the parser's output in order to get the correct struct. The parser will output a `fail` struct if the input is not

syntactically valid XML; in this case, our function should of course return a `fail` struct. However, the parser may return an `ans` struct for something that we don't want to accept as a playlist. We need to determine what types of `ans` structs represent a valid playlist parse; you should look carefully at the format of the output from the parsers and understand how it reflects the structure of particular XML inputs.

Finally, as for optional fields, (artist, album, and picture), remember that not all entries will contain these fields in the XML. Therefore, empty fields (i.e., a certain XML entry doesn't contain an artist field) should have the empty string "" in the corresponding field in the struct that we return (but this is not cause for a `fail`).

So, with those facts in mind, let's write a function that takes the output of a parser (i.e., either an `ans` or a `fail`), and returns either a `playlist` or a `fail`. Because this function is long, complicated, and mostly unenlightening,⁴ we've written most of it for you; there are only a few blanks you'll need to fill in.

Exercise 6.*15 points*

(a) [15 points] Uncomment the code below and fill in the blanks so that `xmlstring->playlist` works correctly. (As above, each group of blanks represents a single combinator or other atom.) **Also write a brief, one-sentence comment above each definition describing its meaning.** For some of the functions, we've annotated the types of arguments to help you (but you still need to write comments).

```

; r is an ans struct
(define (complete-parse? r)
  (and (not (fail? r))
       (empty? (ans-unconsumed r))))

; p is a parser combinator, s is a string
(define (matches? p s)
  (complete-parse? (___ (string->list s))))

; name is a string, field is an xfield struct
(define (valid-nested-field? name field)
  (and (equal? name (xfield-name field))
       (list? (xfield-body field))))

; name and field as above, p is a parser combinator
(define (valid-base-field? name field p)
  (and (equal? name (xfield-name field))
       (string? (xfield-body field))
       (matches? p (xfield-body field))))

```

⁴Not because we're writing it badly, but because (by necessity) the playlist specification itself is long, complicated, and mostly unenlightening! (It could be far worse: cf. <http://www.ex-parrot.com/~pdw/Mail-RFC822-Address.html>) So, if you're tasked with designing a specification in the Real World, imagine you're the one who has to implement it, and design accordingly!

```

; x is the result of a parse
(define (parse->playlist x)
  (if (or (not (complete-parse? x))
        (not (valid-nested-field? "playlist" (ans-result x))))
      (___)
      (foldr add-entry-parse empty-playlist (xfield-body (ans-result x)))))

; p is a predicate, l is a list
(define (exists? p l)
  (and (cons? l) (or (p (car l)) (exists? p (cdr l)))))

; x is of any type, l is a list
(define (in? x l)
  (exists? (lambda (y) (equal? x y)) l))

; a is a string
(define (attr-mandatory? a)
  (in? a (list "name" "filename" "artist")))

(define field-specs
  (list (list "name" (star (alt alpha white)))
        (list "filename"
              (cat (plus (cat (star (alts (list alpha digit))) (char #\.)
                            (str "mp3"))))
                    (list "artist" (star (alt alpha white)))
                    (list "album" (star (alt alpha white)))
                    (list "picture"
                          (cat (plus (cat (star (alts (list alpha digit))) (char #\.)
                                          (alt (str "png") (str "jpg")))))))))

; e-attrs is a list of xfield structs
(define (get-attrs-in e-attrs)
  (lambda (attr-spec)
    (let* ([vals (map xfield-body
                     (filter
                      (lambda (x) (valid-base-field?
                                   (first attr-spec) x (second attr-spec)))
                      e-attrs))]
           [l (length vals)])
      (cond [(= l 0) (if (attr-mandatory? (first attr-spec)) (make-fail) "")]
            [(= l 1) (car vals)]
            [else (make-fail)])))) ; attributes may occur only once in an entry

; e-result is an entry-result from a parse, and is an xfield struct
; p is a playlist
(define (add-entry-parse e-result p)

```

```

(if (or (fail? p)
      (not (valid-nested-field? "entry" e-result)))
    (make-fail)
    (let ([r (map (get-attrs-in (xfield-body e-result)) field-specs)])
      (if (exists? fail? ___)
          (make-fail)
          (playlist-add-entry (apply make-entry r) p))))))

(define xmlstring->playlist (compose ___ xml ___))

; The following are for debugging purposes (you need not comment these).

(define (entry->string e)
  (string-append (entry-name e) ",_by_" (entry-artist e)
                 (if (equal? "" (entry-album e)) ""
                     (string-append "_[\" (entry-album e) \"]"))
                 ":_\" (entry-filename e)
                 (if (equal? "" (entry-picture e)) ""
                     (string-append "_,_" (entry-picture e))))))

(define (playlist->string p)
  (if (fail? p) "Faillist!\n"
      (apply string-append
              "Playlist:\n" (map (lambda (e)
                                (string-append (entry->string e) "\n"))
                               (playlist-entries p)))))

(define print-playlist
  (compose
   display ; Avert your eyes, etc.
   playlist->string))

(define (xfield->string x)
  (string-append "[" (xfield-name x) ":_\"
                 (if (list? (xfield-body x))
                     (string-append "("
                                     (apply string-append (map xfield->string (xfield-body x)))
                                     ")")
                     (xfield-body x)) "]""))

(define (xparse->string p)
  (if (fail? p) "ParseFail!"
      (string-append (xfield->string (ans-result p))
                     "_{Rem:_" (list->string (ans-unconsumed p)) "}")

(define (print-xparse p)
  (display (string-append (xparse->string p) "\n")))

```

```
(define (parse-print parser xs)
  (print-xparse (parser (string->list xs))))
```

6 Now What?

Now that we've parsed our playlists from XML into a nice, machine-friendly form (Scheme structs),⁵ we can process them however we want. We saw above how easy it was to pretty-print the playlists; we could also, e.g., sort them by name or by artist, or find all songs by a given artist, or even write our own Scheme-flavored music player!⁶

7 Finishing Up

(1 point total)

Exercise 7.

1 point

(a) [1 point]

Please tell us how much time you spent on this problem set. You may use this space to add any thoughts or questions you want to send to your TF as well, if you so desire.

⁵XML, of course, has the rare advantage of being neither machine-friendly nor human-friendly! ;-)

⁶iCons?