CSC 526 - Principles of Compiler Design II

Parser Combinators

Overview

- Parser combinators
- Foundation: a simple parser
- Basic parser combinators
- Building combinators from other combinators

Parser Combinators

- A parser combinator is a higher order function that accepts several parsers as input and returns a new parser as its output
- A parser combinator library can be written in your favorite programming language
- Parser combinators do not require any preprocessing; basically do all of lexing, parsing, and AST transformation

A Simple Parser (Version 1)

Parse the character 'A'

```
let pcharA (input : string) : bool * string =
  if (String.length str) = 0 then
    let msg = "No more input" in
    (false, "")
  else
    if input.[0] = 'A' then
      let remaining = String.sub str 1 ((String.length str)
      (true, remaining)
    else
      (false, input)
```

A Simple Parser (Version 2)

Parse any character

```
let pchar (charToMatch : char) (str : string) : (char * str
  if (String.length str) = 0 then
    let msg = "No more input" in
    (msg, "")
  else
    let first = str.[0] in
    if first = charToMatch then
      let remaining = String.sub str 1 ((String.length str)
      let msg = sprintf "Found %c" charToMatch in
      (msg, remaining)
    else
      let msg = sprintf "Expecting '%c'. Got '%c'" charToMa
      (msg, str)
```

A Simple Parser (Version 3)

Make a return type to indicate success or failure

```
type 'a result =
  | Success of 'a
  | Failure of string
```

A Simple Parser (Version 3)

Make a return type to indicate success or failure

```
let pchar (charToMatch : char) (str : string)  : (char * st
  if (String.length str) = 0 then
    Failure "No more input"
  else
    let first = str.[0] in
    if first = charToMatch then
      let remaining = String.sub str 1 ((String.length str)
      Success (charToMatch, remaining)
    else
      let msg = sprintf "Expecting '%c'. Got '%c'" charToMa
      Failure msg
```

A Simple Parser (Version 4)

Return a function

```
let pchar (charToMatch : char) : string -> (char * string) 
  let inner_fun str =
    if (String.length str) = 0 then
      Failure "No more input"
    else
      let first = str.[0] in
      if first = charToMatch then
        let remaining = String.sub str 1 ((String.length st
        Success (charToMatch, remaining)
      else
        let msg = sprintf "Expecting '%c'. Got '%c'" charTo
        Failure msg
  in inner_fun
```

A Simple Parser (Version 5)

Wrap the function in a type

```
type 'a parser = Parser of (string -> ('a * string) result)

let pchar (charToMatch : char) : char parser =
  let inner_fun str =
    if (String.length str) = 0 then
      Failure "No more input"
    else
      let first = str.[0] in
      if first = charToMatch then
        let remaining = String.sub str 1 ((String.length st
        Success (charToMatch, remaining)
      else
        let msg = sprintf "Expecting '%c'. Got '%c'" charTo
        Failure msg
  in Parser inner_fun
```

A Simple Parser (Version 5)

Running a parser:

```
type 'a result =
  | Success of 'a
  | Failure of string

type 'a parser =
  Parser of (string -> ('a * string) result)

let run (p : 'a parser) (input : string) : ('a * string) re
  let Parser(inner_fun) = p in
  inner_fun input
```

Combinators

- A combinator library is designed around combining things to get more complex values of the same type
- Examples:
  - integer + integer = integer
  - list @ list = list
  - parser ?? parser = parser (what should ?? be?)

Basic Parser Combinators

- parser andThen parser -¿ parser
- parser orElse parser -¿ parser
- parser map (transformer) -¿ parser

AndThen Parser Combinator

- Run the first parser
  - If there is a failure, then return
- Otherwise, run the second parser with the remaining input
  - If there is a failure, then return
- If both parsers succeed, then return a tuple that contains both parsed values

AndThen Parser Combinator

```
let andThen (p1 : 'a parser) (p2 : 'b parser) : ('a * 'b) p
  let inner_fun input =
    (* run the first parser *)
    let result1 = run p1 input in
    (* test for Failure/Success *)
    match result1 with
    | Failure err -> Failure err (* return error from p1 *)
    | Success (value1, remaining1) ->
      (* run the second parser with the remaining input *)
      let result2 = run p2 remaining1 in
      (* test for Failure/Success *)
      match result2 with
      | Failure err -> Failure err
      | Success (value2, remaining2) ->
          Success( (value1, value2), remaining2)
  in Parser inner_fun
```

OrElse Parser Combinator

- Run the first parser
  - On success, return the parsed value along with the remaining input
- Otherwise, on failure, run the second parser with the original input
- Return the result of the second parser

OrElse Parser Combinator

```
let orElse (p1 : 'a parser) (p2 : 'a parser) : 'a parser =
  let inner_fun input =
    (* run the first parser *)
    let result1 = run p1 input in
    (* test for Failure/Success *)
    match result1 with
    | Success result -> result1
    | Failure err -> run p2 input
  in Parser inner_fun
```

Map Parser Combinator

- Run the parser
- On success, transform the parsed value using the provided function
- Otherwise, return failure

Map Parser Combinator

```
let mapP (f : 'a -> 'b) (p : 'a parser) : 'b parser =
  let inner_fun input =
    (* run the parser on the input *)
    let result = run p input in
    (* match result for Failure/Success *)
    match result with
    | Success (value, remaining) -> Success(f value, remain
    | Failure err -> Failure err
  in Parser inner_fun
```

Parser Combinator Operators

- andThen: `pcharA *>>* pcharB`
- orElse: `pcharA <|> pcharB`
- mapP: `pcharA |>> (...)`

Combining Basic Parser Combinators

```
(* parse any char from a list *)
let choice (ps : 'a parser list) : 'a parser =
  match ps with
  | x::xs -> List.fold_left ( <|> ) x xs
  | _ -> failwith "Empty list to choice"


(* parse any char from a list *)
let anyOf (cs : char list) : char parser =
  cs |> List.map pchar |> choice


(* parse a digit *)
let parseDigit = anyOf ['0';'1';'2';'3';'4';'5';'6';'7';'8'
```

Combining Basic Parser Combinators

```
(* convert a list of parsers to a parser of a list *)
let rec sequence (ps : 'a parser list) : 'a list parser =
  let cons head tail = head::tail in
  let consP = lift2 cons in
  match ps with
  | [] -> returnP []
  | head::tail -> consP head (sequence tail)
```

Combining Basic Parser Combinators

```
(* helper function to create a string from a list of charac
let string_of_char_list (cs : char list) : string =
  List.fold_left (^) "" (List.map (fun c -> String.make 1 c

(* parse a string *)
let pstring (str : string) : string parser=
  str
  |> String.to_seq
  |> List.of_seq
  |> List.map pchar
  |> sequence
  |> mapP string_of_char_list
```

"More than one" combinators

```
let many p = ...   (* zero or more *)
let many1 p = ... (* one or more *)
let opt p = ...    (* zero or one *)


(* parse whitespace *)
let whitespaceChar : char parser = anyOf [' '; '\t'; '\n']
let whitespace : char list parser = many whitespaceChar
```

"Throwing away" combinators

```
p1 *>> p2  (* throw away right side *)
p1 >>* p2  (* throw away left side*)

(* keep only the inside value *)
let between p1 p2 p3 =
  p1 >>* p2 *>> p3

(* parse integers between quotes *)
let pdoublequote = pchar '"'
let quotedint = between pdoublequote pint pdoublequote
```

"Separator" combinators

```
let sepBy1 p sep = ... (* one or more p separated by sep *)
let sepBy p sep = ...  (* zero or more p separated by sep *)

(* example *)
let comma = pchar ','
let digit = anyOf ['0';'1';'2';'3';'4';'5';'6';'7';'8';'9']
let oneOrMoreDigitList = sepBy1 digit comma
```