

Functional Programming

WS 2007/08

Christian Sternagel¹ (VO + PS)
Friedrich Neurauter² (PS)
Harald Zankl³ (PS)

Computational Logic
Institute of Computer Science
University of Innsbruck

14 December 2007

¹christian.sternagel@uibk.ac.at

²friedrich.neurauter@uibk.ac.at

³harald.zankl@uibk.ac.at

Overview

Week 9 - Combinator Parsing

- Summary of Week 8

- Motivation

- Combinator Parsing

- An Example Parser

Overview

Week 9 - Combinator Parsing

Summary of Week 8

Motivation

Combinator Parsing

An Example Parser

Efficiency of Functional Programs

Avoid unnecessary recomputations by ...

- ▶ tupling

Introduce tail recursion by ...

- ▶ parameter accumulation

Overview

Week 9 - Combinator Parsing

Summary of Week 8

Motivation

Combinator Parsing

An Example Parser

What is Parsing?

Parsing is the decomposition of a *linear sequence* into a *structure*, given by a *grammar*. The linear sequence may be a text in some natural language, a computer program, a web site, a piece of music, a sequence of genes, . . .

In the Following ...

Use

- ▶ linear sequence: **l-string** (i.e., **char list**)
- ▶ structure: some user-defined type
- ▶ grammar: BNF (Backus-Naur form)

Note

- ▶ BNF can express context-free grammars (CFG)
- ▶ combinator parsers can parse context-sensitive grammars
- ▶ however, for the purpose of this lecture CFG suffice

Example: Arithmetic Expressions

Grammar

```

e ::= e + t | t
t ::= t * f | f
f ::= ( e ) | n
n ::= ε | d n
d ::= 0 | ... | 9

```

e ... “expression”
t ... “term”
f ... “factor”
n ... “natural number”
d ... “digit”

Structure - Abstract Syntax Tree (AST)

```

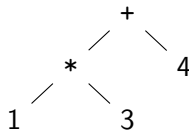
type arith =
  | Num of Strng.t
  | Add of arith * arith
  | Mul of arith * arith
;;

```

Example

► input: "1*3+4"

► AST:



Parsers

First Attempt

- ▶ functions of type `Strng.t -> ('a * Strng.t)`
- ▶ e.g., `digit ['1'; '2']` results in `('a', ['2'])`
- ▶ but there is information missing, e.g.: *error?* or *input consumed?*

Type of parsers

```
type 'a result;;
```

```
type 'a t = Strng.t -> 'a result;;
```

- ▶ `result` is an abstract data type (ADT)
- ▶ the only thing known about parsers is that they take an l-string as input and return *some* result

Overview

Week 9 - Combinator Parsing

Summary of Week 8

Motivation

Combinator Parsing

An Example Parser

The Internal Representation

Type

```
type 'a reply = Error | Ok of ('a * Strng.t);;  
type 'a consumed = Empty of 'a | Consumed of 'a;;  
type 'a result = ('a reply) consumed;;  
type 'a t = Strng.t -> 'a result;;
```

- ▶ **reply** keeps record whether error occurred or not
- ▶ **consumed** keeps record whether input has been consumed
- ▶ **result** combines both pieces of information
- ▶ concrete type only visible within **Parser** module

A Framework for Using Parsers

Checking for errors

```
let reply = function
  | Empty r -> r
  | Consumed r -> r
;;
```

```
type 'a reply = Error | Ok of ('a * Strng.t);;
type 'a consumed = Empty of 'a | Consumed of 'a;;
type 'a result = ('a reply) consumed;;
type 'a t = Strng.t -> 'a result;;
```

Applying a Parser

```
let parse p s = match reply (p s) with
  | Error -> failwith "Parser.parse: no successful parse possible"
  | Ok (x, _) -> x
;;
```

For Convenience

```
let test p s = parse p (Strng.of_string s);;
```

Primitive Parsers

Sat



```
let sat p = function
  | [] -> Empty Error
  | c :: cs -> if p c then Consumed (Ok (c, cs)) else Empty Error
;;
```

- ▶ only primitive parser
- ▶ `sat p` accepts any character for which `p` is **true**

Example

```
# sat ((<>) 't') ['a'; 't'];;
- : (char reply) consumed = Consumed (Ok ('a', ['t']))
```

Primitive Parsers (cont'd)

Any char



```
let any_char = sat (fun _ -> true);;
```

- ▶ accepts any single character

Character



```
let char c = sat ((=) c);;
```

- ▶ accepts only the given character `c`

Parser Combinators

Bind

```
let (>>=) p f = (fun s -> match p s with
| Consumed r -> begin match r with
| Ok (x, s') -> begin match f x s' with
| Empty r' -> Consumed r'
| consumed -> consumed
end
| Error -> Consumed Error
end
| Empty r -> begin match r with
| Ok (x, s') -> f x s'
| Error -> Empty Error
end
);;
```

Parser Combinators (cont'd)

Then

```
let (>>) p q = p >>= fun _ -> q;;
```

Example

- ▶ $p ::= a b$
- ▶ `let p = char 'a' >>= fun _ -> char 'b';;`
- ▶ `let p = char 'a' >> char 'b';;`
- ▶ i.e., (`>>=`) and (`>>`) correspond to juxtaposition in BNF
- ▶ (`>>=`) is used if result matters
- ▶ (`>>`) is used otherwise

Parser Combinators (cont'd)

Choice

```
let (<|>) p q = (fun s -> match p s with  
  | Empty Error -> q s  
  | other -> other  
);
```

Example

- ▶ $p ::= a \mid b$
- ▶ **let** p = char 'a' <|> char 'b';;
- ▶ i.e., (<|>) corresponds to | in BNF

Parser Combinators (cont'd)

Return

```
let return x = fun s -> Empty (Ok (x, s));;
```

Example

```
let any_pair =  
  any_char >>= fun l ->  
  any_char >>= fun r ->  
  return (l, r)  
;;
```

Parser Combinators (cont'd)

Many

- ▶ **many** p applies p zero or more times
- ▶ result is list of results of p
- ▶ greedy (as many applications of p as possible)

Example

- ▶ $p ::= \epsilon \mid a p$
- ▶ **let** $p = \text{many} (\text{char 'a'})$

Overview

Week 9 - Combinator Parsing

Summary of Week 8

Motivation

Combinator Parsing

An Example Parser

Recall

Grammar

$$e ::= e + t \mid t$$
$$t ::= t * f \mid f$$
$$f ::= (e) \mid n$$
$$n ::= \epsilon \mid d n$$
$$d ::= 0 \mid \dots \mid 9$$

Structure (Abstract Syntax Tree)

```
type arith =  
  | Num of Strng.t  
  | Add of arith * arith  
  | Mul of arith * arith  
;;
```

The First Attempt

```
let rec e x = (  
  (e >>= fun e1 -> char '+' >> t >>= fun e2 -> return (Add (e1, e2)))  
  <|> t  
) x  
and t x = (  
  (t >>= fun t1 -> char '*' >> f >>= fun t2 -> return (Mul (t1, t2)))  
  <|> f  
) x  
and f x = (  
  (char '(' >> e >>= fun e -> char ')') >> return e)  
  <|> n  
) x  
and n = many1 digit >>= fun r -> return (Num r);;
```

Problem

left recursion

Solution: Eliminate Left Recursion

Revised Grammar

$$e ::= t e'$$
$$e' ::= + t e' \mid \epsilon$$
$$t ::= f t'$$
$$t' ::= * f t' \mid \epsilon$$
$$f ::= (e) \mid n$$
$$n ::= \epsilon \mid d n$$
$$d ::= 0 \mid \dots \mid 9$$

The Final Parser

```
let rec e x = (t >>= e') x
and e' l =
  (char '+' >> t >>= e' >>= fun r -> return (Add (l, r)))
  <|> return l
and t x = (f >>= t') x
and t' l =
  (char '*' >> f >>= t' >>= fun r -> return (Mul (l, r)))
  <|> return l
and f x =
  ((char '(' >> e >>= fun r -> char ')') >> return r) <|> n) x
and n = many1 digit >>= fun r -> return (Num r);;
```