Functional Programming

WS 2008/09

Christian Sternagel (VO)[1] Friedrich Neurauter (PS)[2]

Computational Logic
Institute of Computer Science
University of Innsbruck

28 November 2008

[1] christian.sternagel@uibk.ac.at
[2] friedrich.neurauter@uibk.ac.at

Efficiency of Functional Programs

Avoid unnecessary recomputations by . . .

- tupling

Introduce tail recursion by . . .

- parameter accumulation

This Week

Practice I

OCaml introduction, lists, strings, trees

Theory I

lambda-calculus, evaluation strategies, induction, reasoning about functional programs

Practice II

efficiency, tail-recursion, combinator-parsing

Theory II

type checking, type inference

Advanced Topics

lazy evaluation, infinite data structures, monads, ...

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

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

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

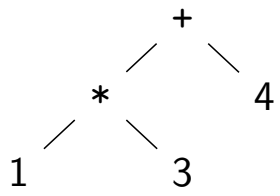
Structure - Abstract Syntax Tree (AST)

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

Example: Arithmetic Expressions (cont'd)

Example

- input: `"1*3+4"`
- AST:



Remark

AST contains more information than input

Parsers

First Attempt

- functions of type `Strng.t -> ('a * Strng.t)`
- e.g., `digit ['1';'2']` results in `('1',['2'])`
- but what about errors?

Type of parsers

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

The Internal Representation

Type

```
type input = Strng.t
type 'a result = ('a * input)option
type 'a t = input -> 'a result
```

- on success `Some(x,i)` is returned, where `x` is the result and `i` the remaining input
- on error `None` is returned
- concrete type only visible within `Parser` module

A Framework for Using Parsers

Applying a Parser

```
let parse p s = match p (Strng.of_string s) with
  | None       -> None
  | Some(x,_) -> Some x
```

For Convenience

```
let test p s = match p (Strng.of_string s) with
  | None       -> failwith "no␣parse␣possible"
  | Some(x,_) -> x
```

Primitive Parsers

Sat

```
let sat p = function []    -> None
                   | c::cs -> if p c then Some(c,cs)
                                     else None
```

- `sat p` accepts any character for which `p` is **`true`**

Example

```
# sat ((<>) 't') ['a';'t'];;
- : char Parser.result = Some ('a', ['t'])
```


Primitive Parsers (cont'd)

Eoi

```
let eoi = function [] -> Some((),[])
                  | _  -> None
```

- accepts if end of input is reached
- useful to ensure that full input has been consumed

Example

what is the difference between

```
let p = sat (fun c -> c = 'a')
```

and

```
let p' = sat (fun c -> c = 'a') >> eoi
```

Character Parsers

Any

```
let any = sat (fun _ -> true)
```

- accepts any single character

Char

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

- accepts only the given character `c`

Parser Combinators

Bind

```
let (>>=) p f i = match p i with None      -> None
                               | Some(x,i) -> f x i
```

Then

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

Example: Parser Combinators

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 i = match p i with None        -> q i
                               | Some _ as r -> r
```

Example

- $p ::=$ a $|$ b
- `let p = char 'a' <|> char 'b'`
- i.e., `(<|>)` corresponds to $|$ in BNF

Parser Combinators (cont'd)

Return

```
let return x = fun i -> Some(x,i)
```

Example

```
let any_pair =
 any >>= fun l ->
 any >>= 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

- $p ::= \epsilon \mid \mathrm{a}\ p$
- `let p = many (char 'a')`

Recall

Grammar

$$
\begin{aligned}
e &::= e + t \mid t \\
t &::= t * f \mid f \\
f &::= (e) \mid n \\
n &::= d \mid d\ n \\
d &::= 0 \mid \ldots \mid 9
\end{aligned}
$$

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 = (
 between (char '(') e (char ')')
 <|> n
) x
and n = many1 digit >>= fun r -> return(Num r)
```

Problem

left recursion

Solution: Eliminate Left Recursion

Revised Grammar

$$
\begin{aligned}
e &::= t\ e' \\
e' &::= +\ t\ e' \mid \epsilon \\
t &::= f\ t' \\
t' &::= *\ f\ t' \mid \epsilon \\
f &::= (\ e\ ) \mid n \\
n &::= d \mid d\ n \\
d &::= 0 \mid \ldots \mid 9
\end{aligned}
$$

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 = (between (char '(') e (char ')') <|> n) x
and n = many1 digit >>= fun r -> return(Num r)
```