Case Study: Parsing

- Recursive Descent Parsing.
- Parsing functionals that closely correspond to the rules of a grammar.
- Functionals that correspond to concatenation, alternatives, repetition, etc.
- Easy to allow backtracking.
- Contruct syntax trees to which it is easy to attach semantics.
- Similar functionals in other domains.

caveat: The functions are easy to write - the parsing algorithms are the same.

Outline

A lexical analyser converts the string of symbolsl into a list of tokens.

```
datatype token = Key of string
               | Id of string;
```

`Key` for keywords, and `Id` for identifiers.

A parser is a function of type:

```
type 'a phrase =
     token list -> 'a * token list;
```

The parser removes tokens from the front of the list that match the rule of type `'a`.

Basic Parsers

```
$"string" : string phrase

id : string phrase

empty : 'a list phrase
```

```
exception Error of string;

fun id (Id a::toks) = (a, toks)
  | id toks =
     raise Error "Identifier expected";

fun empty toks = ([], toks);
```

Alternatives and Consecutives

```
infix 0 || ;

fun (ph1 || ph2) toks =
  ph1 toks  handle Error _ => ph2 toks;


infix 5 -- ;

fun (ph1 -- ph2) toks =
  let val (x, toks2) = ph1 toks
      val (y, toks3) = ph2 toks2
  in  ((x,y), toks) end;
```

Transformation and Repetition

```
infix 3 >> ;

fun (ph >> f) toks =
   let val (x, toks2) = ph toks
   in  (f x, toks2) end;


fun repeat ph toks =
    (ph -- repeat ph >> (op::)
     || empty) toks;
```

Propositional Logic 1

Grammar rules:

$$\begin{aligned} \mathit{Prop} &= \mathit{Term}\{\texttt{"|"}\ \mathit{Term}\} \\ \mathit{Term} &= \mathit{Factor}\{\texttt{"\&"}\ \mathit{Factor}\} \\ \mathit{Factor} &= \mathit{Id} \mid \texttt{"\textasciitilde"}\ \mathit{Factor} \mid \texttt{"("}\ \mathit{Prop}\texttt{")"} \end{aligned}$$

```
fun orl(p, []) = p
  | orl(p,(_,q)::l) = orl(Disj(p,q), l);

fun andl(p, []) = p
  | andl(p,(_,q)::l) = orl(Conj(p,q), l);
```

Propositional Logic 2

```
fun prop toks =
  (term -- repeat($"|" -- term)
                    >> orl) toks;

fun term toks =
  (factor -- repeat($"&" -- factor)
                       >> andl) toks;

fun factor toks =
   (   id >> Atom
    || $"~" -- factor
          >> (fn (_,p) => Neg p)
    || $"(" -- prop -- $")"
          >> (fn ((_,p),_) => p)
    ) toks;
```

Backtracking Parser

The parser yields 1 outcome, or an error.

We can obtain a backtracking parser by replacing the exception by a lazy list of outcomes.

```
[]
[(x, toks)]
[(x1, toks1), (x2, toks2),...]
```

Depending on whether the string of tokens is a syntax error, a unique parse, or an ambiguous parse.

`--` and `||` can be modified to combine lazy lists.

The lazy consumption of outcomes leads to backtracking.