Praktische Programmierung mit ML

Fallbeispiel: Lexikalische Analyse und Parser

Lexikalische Analyse

Zeichensequenz $\rightarrow$ Tokensequenz

```
(* Charakterisierung einzelner Zeichen (hier: einelementige Strings) *)

- fun IsDigit x="0" <= x andalso x <= "9";
val IsDigit = fn : string -> bool

- fun IsLetter x= ("a" <= x andalso x<="z") orelse ("A"<=x andalso x<="Z");
val IsLetter = fn : string -> bool

- fun IsSeparator x= (x=" " orelse x ="\n" orelse x="\t");
val IsSeparator = fn : string -> bool
```


Zahl erkennen

```
- fun GetNumAux buf [] = (concat(rev buf), [])
        | GetNumAux buf (l as (x::l1)) =
                if IsDigit x then GetNumAux (x::buf) l1
                        else (concat(rev buf), l);

- GetNumAux ["a","b","c"] ["1", "2", "3"," ","4","5"];
val it = ("cba123",[" ","4","5"]) : string * string list
```

```
- val GetNum = GetNumAux [];
val GetNum = fn : string list -> string * string list

- GetNum ["1","2","3"," ","4","5"];
val it = ("123",[" ","4","5"]) : string * string list
```

Bezeichner erkennen

```
- fun GetIdentAux buf [] = (concat(rev buf), [])
      | GetIdentAux buf (l as (x::l1)) =  if IsLetter x orelse IsDigit x
                                          then GetIdentAux (x::buf) l1
                                          else (concat(rev buf), l);
val GetIdent = fn : string list -> string * string list

- GetIdentAux ["a","b","c"] ["e","f","g","4","5"," ","6","7"];
val it = ("cbaefg45",[" ","6","7"]) : string * string list
```

```
- exception GetIdentErr;
exception GetIdentErr
Warning: match nonexhaustive  x :: l => ...

- fun GetIdent (x::l) =
      if IsLetter x then GetIdentAux [x] l else raise GetIdentErr;
val GetIdent = fn : string list -> string * string list
```



```
- fun GetTail p buf []            = (concat(rev buf), [])
    | GetTail p buf (l as x::l1) =
                  if p x then GetTail p (x::buf) l1 else (concat(rev buf), l);
val GetTail = fn  : (string -> bool) -> string list -> string list -> string * string
list

- fun GetNextToken [x] = (x, [])
      | GetNextToken (x::l) =
              if IsLetter x
              then GetTail (fn x=>IsLetter x orelse IsDigit x) [x] l
              else if IsDigit x
                   then GetTail IsDigit [x] l
                   else (x, l);
val GetNextToken = fn : string list -> string * string list

- fun Tokenise [] = []
      | Tokenise (l as x::l1) =
              if IsSeparator x  then Tokenise l1
                                else let val (t,l2) = GetNextToken l
                                     in t::(Tokenise l2) end;
val Tokenise = fn : string list -> string list
```



```
- fun RconcatAux [] = [] | RconcatAux (x::lr) = str(x)::(RconcatAux lr);
val RconcatAux = fn : char list -> string list

- fun Rconcat s = RconcatAux (explode(s));
val Rconcat = fn : string -> string list

- Tokenise (Rconcat "123abcde1][ ] 56a");
val it = ["123","abcde1","]","[","]","56","a"] : string list
```

Mehr-Zeichen-Symbole

<=, <<, =>, =, ==>, ->

```
- val SpecTab = [("=", ["<", ">", "="]),
                 ("<", ["<", ">"]),
                 (">", ["<", ">"]),
                 ("==", [">"])];

- fun Mem x [] = false     | Mem x (x1::l) = (x=x1) orelse Mem x l;

- fun Get x [] = [] | Get x ((x1,l)::rest) = if x=x1 then l else Get x rest;
- Get "=" SpecTab;
val it = ["<",">","="] : string list

- fun GetSymbol spectab tok [] = (tok,[])
    | GetSymbol spectab tok (l as x::l1) =
        if Mem x (Get tok spectab) then GetSymbol spectab (tok^x) l1
                                   else (tok, l);
```



```
- fun GetNextToken spectab [x] = (x, [])
    | GetNextToken spectab (x::(l as x1::l1)) =
        if IsLetter x
        then GetTail (fn x=>IsLetter x orelse IsDigit x) [x] l
        else if IsDigit x
             then GetTail IsDigit [x] l
             else if IsDigit x
                  then GetTail IsDigit [x] l
                  else if Mem x1 (Get x spectab)
                       then GetSymbol spectab (concat[x, x1]) l1
                       else (x, l);
- fun Tokenise spectab [] = []
      | Tokenise spectab (l as x::l1) =
          if IsSeparator x
          then Tokenise spectab l1
          else let val (t,l2) = GetNextToken spectab l
               in t::(Tokenise spectab l2) end;

- fun Lex s = Tokenise SpecTab (Rconcat(s));
val Lex = fn : string -> string list
```


Parser - rechtsassoziativ

Tokenfolge → Ableitungsbaum

x, f x, **f x y z**, (f x) y, f(f x), f(g x)(h x)...

```
- datatype tree = Atom of string
         | Comb of tree * tree;
datatype tree = Atom of string | Comb of tree * tree

- fun Parse [next] = Atom next
         | Parse (next::rest) = Comb(Atom next, Parse rest);

(* open Compiler.Control.Print; printDepth := 10; *)

- Parse ["f","x","y","z"];
val it = Comb (Atom "f",Comb (Atom "x",Comb (Atom "y",Atom "z"))) : tree
```

Parser - linksassoziativ

```
- fun Parser t [] = t
  | Parser t (next::rest) = Parser (Comb(t, Atom next)) rest;

- fun Parse [next] = Atom next
  | Parse (next::rest) = Parser (Atom next) rest;

- Parse ["f","x","y","z"];
val it = Comb (Comb (Comb (Atom "f",Atom "x"),Atom "y"),Atom "z") : tree
```


Top-Down-Präzedenz Parser

```
- datatype tree = Nil
        | Atom of string
        | Comb of tree * tree
        | Node of string * tree list;
datatype tree
  = Atom of string | Comb of tree * tree | Nil | Node of string * tree list
```

```
- exception NilRightArg;

- fun MkComb(Nil,t2) = t2
        | MkComb(t1,Nil) = raise NilRightArg
        | MkComb(t1, t2) = Comb(t1, t2);
val MKComb = fn : tree * tree -> tree
```


Vorbereitungen

```
SymTab "+" = (6, ParseSeq (....) 5 [Id])
SymTab x = (10, ParseAtom);
....
```

```
type parser = int -> tree -> string list -> tree * string list;

type symtab = string -> int * (parser -> parser);

- fun Parser symtab (m:int) t [] = (t, [])
        | Parser symtab m t (inp as next::rest) =
          let val (n, parsefn) = symtab next
          in if m>=n then (t, inp)
                  else parsefn (Parser symtab) m t inp
          end;
val Parser = fn
 : ('a -> int * ((int -> 'b -> 'a list -> 'b * 'a list)
        -> int -> 'b -> 'a list -> 'b * 'a list))
   -> int -> 'b -> 'a list -> 'b * 'a list
```

```
- fun Parser symtab (m:int) t [] = (t, [])
        | Parser symtab m t (inp as next::rest) =
          let val (n, parsefn) = symtab next
          in if m>=n then (t, inp)
                  else parsefn (Parser symtab) m t inp
          end;
- fun Parser (tab:symtab) : parser =
                fn m =>
                    fn t =>
                      fn [] => (t, [])
        |               (inp as next::rest) =>
          let val (n, parsefn) = tab next
          in if m>=n then (t, inp)
                  else parsefn (Parser tab) m t inp
              end;
val Parser = fn : symtab -> parser
```


Funktionen für die Symboltabelle

```
fun ParseAtom parse p t (next::rest) =
        parse p (MkComb(t, Atom next)) rest;

exception MissingClosingBracket;

fun ParseBracket close parse p t (_::rest) =
        let val (t', next'::rest') = parse 0 Nil rest
        in if close=next' then parse p (MkComb(t,t')) rest'
                    else raise MissingClosingBracket
        end;

exception TerminatorParseErr;

fun Terminator parse _ = raise TerminatorParseErr;
```


Idee: bestimme rekursiv für alle Bestandteile eines Konstrukts die Teilbäume und füge diese zusammen

Bsp. if e then e_1 else e_2

```
fun ParseSeq mktree m (fl: (parser->parser) list) (parse:parser) : parser =
    fn n => fn t => fn (_::rest) =>
     let val (l, rest1) = ParseSeqAux m fl parse n rest
     in parse n (mktree(t,l)) rest1 end;
```

```
fun Id x = x;

exception ChkAftErr;
fun ChkAft s parse m t inp = case parse m t inp
    of (t', s'::rest) => if s=s' then (t', rest)
                    else raise ChkAftErr;

fun MkUnop unop (t,tl) = MkComb(t, Node(unop, tl));
fun MkBinop bnop (t,tl) = Node(bnop, t::tl);
```

Die Ableitungsfunktion

```
fun ParseSeqAux m [f:parser->parser] (parse:parser) n inp =
        let val (t, rest1) = f parse m Nil inp
        in ([t], rest1) end
  | ParseSeqAux m (f::fl : (parser->parser)list) parse n inp =
        let val (t, rest1) = f parse 0 Nil inp
        in let val (l,rest2) = ParseSeqAux m fl parse 0 rest1
                in (t::l, rest2) end
        end;

fun ParseSeq mktree m (fl: (parser->parser)list) (parse:parser) : parser =
        fn n => fn t => fn (_::rest) =>
         let val (l, rest1) = ParseSeqAux m fl parse n rest
         in parse n (mktree(t,l)) rest1 end;
```


Die Symboltabelle

```
fun SymTab "*" = (7, ParseSeq (MkBinop "MULT") 8 [Id])
  | SymTab "+" = (6, ParseSeq (MkBinop "ADD") 5 [Id])
  | SymTab "~" = (10, ParseSeq (MkUnop "MINUS") 9 [Id])
  | SymTab "if" = (10, ParseSeq (MkUnop "COND") 5 [ChkAft "then",
                                                   ChkAft "else",
                                                   Id ])
  | SymTab "(" = (10, ParseBracket ")")
  | SymTab ")" = (0, Terminator)
  | SymTab "then" = (0, Terminator)
  | SymTab "else" = (0, Terminator)
  | SymTab x = (10, ParseAtom);

use("Lex.txt");
fun P s = Parser SymTab 0 Nil (Lex s)
```


„Fröhliches Parsen“

```
- open Compiler.Control.Print;
opening Compiler.Control.Print
  val printDepth : int ref
  ....

- printDepth :=15;
val it = () : unit

- P "f if x then y + z else y * z";
val it =
  (Comb
    (Atom "f",
     Node
      ("COND",
       [Atom "x",Node ("ADD",[Atom "y",Atom "z"]),
        Node ("MULT",[Atom "y",Atom "z"])])),[]) : tree * string list
```