FemtoML

Florian Müller

30.06.2016

FemtoML

- “Minimale Teilmenge von ML”
- Implementation mit FParsec

Ziel

```
let fibPreMul = 10;;

let rec fib =
    fun n ->
        if n < 2 then 1
        else
            fib (n-1) + fib(n-2);;

let fibMul =
    fun f ->
        fun n -> (fib n) * f * fibPreMul;;

fibMul 4 10;;
```

FParsec Parser

- Parser ist eine Funktion der Form

```
type Parser<'Result,'UserState> =
    CharStream<'UserState> -> Reply<'Result>
```

- 'UserState interessiert uns nicht, daraus folgt

```
type Parser<'Result,_> =
    CharStream<_> -> Reply<'Result>
```

- Der Output ist typisiert. z.B.

```
type Parser<int,_> =
    CharStream<_> -> Reply<int>
```

FParsec Parser

- Input `CharStream`
  - Info über zu parsende Zeichen
  - Aktuelle Position
  - für uns nicht interessant
- Input `Reply`
  - Result
  - Error
  - Status
    - Ok
    - Error
    - FatalError

Bsp

```
let someParser = pstring "abcd"
// CharStream<unit> -> Reply<string>
```

- Über die `run` Funktion von FParsec anwenden

```
let parseResult = "abcd" |> run someParser

match parseResult with
| Success(result, _, _) -> ...
| Failure(errorMsg, _, _) -> ...
```

- `parseResult` vom Typ `ParserResult<'a, unit>`
  - Success of 'Result * 'UserState * Position
  - Failure of string * ParseError * 'UserState

Viele Parser schon vorhanden

```
val puint64: Parser<uint64, 'u>
val pfloat: Parser<float, 'u>

val manySatisfy: (char -> bool) -> Parser<string,'u>

val stringReturn: string -> 'a -> Parser<'a, 'u>

// ... und viele mehr
```

FemtoML

- Unser Parser muss einen AST zurückgeben
- Bei uns: Typ `Expr`

BinOp

```
type BinOp = Eq | Lt | Gt | Add | Sub | Mul | Div
```

Expr

```
type Expr =
    | Int of int
    | Ident of string
    | BinOp of Expr * BinOp * Expr
    | Apply of Expr * Expr
    | If of Expr * Expr * Expr
    | Lambda of string * Expr
    | Let of bool * string * Expr * Expr
    | LetStmt of bool * string * Expr
```

Integer, String

```
let pInt = puint32 |>> int .>> spaces
```

```
let str s = pstring s .>> spaces
```

Identifier (Variable, Funktionsname, Argument)

```
let pIdent =
    let idStr = many1Satisfy2L
                    isLetter
                    isAlphaNum
                    "identifier"
                    .>> spaces

    let p state =
        let reply = idStr state
        // assert reply is not a keyword ...
        // reply not in ["if", "else", ...., "let"]

    attempt p .>> spaces
```

OperatorPrecedenceParser

- Erstellen Parser vom Typ `Expr`. Resultat ist vom Typ `Expr`

```
let opp = OperatorPrecedenceParser<Expr, unit, unit>()
//                                 ^^^^

let pExpr = opp.ExpressionParser

opp.TermParser = ?? // Alles, was kein Operator ist
```

Operatoren hinzufügen

```
opp.AddOperator(
    InfixOperator("<", spaces, 1, Associativity.Left,
                  fun f g -> BinOp(f, Lt, g)))

// ...

opp.AddOperator(
    InfixOperator("-", spaces, 50, Associativity.Left,
                  fun f g -> BinOp(f, Sub, g)))
```

- Precendence: `x * y + z > i` -> `((x * y) + z) > i`

TermParser

```
opp.TermParser =
    choice [
        pAtoms // ???
        pIf // if pExpr then pExpr else pExpr
        pLetExpr // let {rec} pExpr in pExpr
        pLambdaExpr // fun pIdent -> pExpr
     ]
```

Atoms

```
let pAtom =
    choice [pInt |>> Int
            pIdent |>> Ident
            between (str "(") (str ")") pExpr]

let pAtoms = many1 pAtom |>> fun fs ->
    List.reduce(fun f g -> Apply(f, g)) fs


// run pAtoms "foobar barfoo 1"
Apply(
    Apply(
        Ident "foobar", Ident "barfoo"),
    Int 1)
```

Eval

```
val eval = Expr -> Value
```

```
type Value =
    | VInt of int
    | VBool of bool
    | VClosure of string * Map<string, Value ref> * Expr
```

```
let eval vars expr =
    match expr with
    | Int i -> VInt i
    | Ident n -> //...
    // ...
```

Eval

- `Int -> VInt`
- `Ident -> Vint, VClosure`
- `Lambda -> VClosure`
- `BinOp -> ...`

Bsp.

```
| Ident var ->
    match Map.tryFind var vars with
        | Some value -> !value
        | None -> failwith "Unknown '%s'" var
```

Bsp. 2

```
| Apply(func, arg) ->
    match eval vars func, eval vars arg with
    | VClosure(var, vars, body), arg ->
        eval (Map.add var (ref arg) vars) body
    | _ -> failwith "Not a function"
```

Terminal-Client

- Input von User einlesen
- Parsen
- Evaluieren
- Resultat ausgeben

Demo

- Demo

Fragen?

- Nope