Moduly a namespace

Kód v F#pu je rozložen do jmenných prostorů a modulů:
- ♣ jmenný prostor může obsahovat typy a moduly
- ♣ modul může obsahovat cokoliv, tj. typy, hodnoty, funkce a další moduly

Pokud kompilujete zdrojový soubor, který neobsahuje ani definici jmenného prostoru ani modulu, je implicitně umístěn do modulu, jehož jméno se shoduje se jménem zdrojového souboru bez koncovky, přičemž první písmeno je převedeno na velké (tj. **module** `Program` pro soubor `program.fs`). Toto pravidlo funguje jenom pro spustitelné soubory, knihovní moduly musí vždy uvést svůj modul a/nebo jmenný prostor.

Soubor `.fs` může definovat buď jeden top–level modul nebo dva, pokaždé je syntaxe trochu jiná:

♣ jeden top–level modul:
Soubor začíná řádkou **module** `[jmenný_prostor.]jméno_modulu` a zbytek souboru, který nemusí být indentován, je umístěn v tomto modulu. Pokud v souboru jsou nějaké další moduly, jsou uvnitř top–level modulu, a definují se jako

```
module Vnitřní =
   ..
```

přičemž indentace vnitřku modulu určuje, kde modul končí.

```
module MyCompany.MySort.Sort
module Helpers =
   ...
   ...
let best_sort xs = ...
```

♣ několik top–level modulů:
Soubor začíná řádkou **namespace** `jmenný_prostor`. Následující typy a moduly jsou v tomto prostoru. Všechny uvedené moduly se definují jako **module** `M =` a indentace určuje jejich rozsah. Navíc je možné na nejvyšší úrovni použít další **namespace** `jmenný prostor`, čímž se přepne jmenný prostor ve zbytku kódu. Jmenné prostory nejsou nijak vnořené, tj. každý je nezávislý.
Speciální jméno **global** značí "vršek" jmenného prostoru, tj. pomocí

```
namespace global
type Tree<'a> = ...
```

můžeme umístit typ `Tree<'a>` přímo do top–level jmenného prostoru.

```
namespace MyCompany.Trees
type Tree<'a> = ...
module Tree =
   let flatten = ...
   ...
   module AsGraph =
      ...

namespace MyCompany.Graphs
type Graph<'a> = ...
module Graph =
```

Pokud používáte kód z jiného modulu nebo jmenného prostoru než z aktuálního, musíte ho plně kvalifikovat, nebo použít klíčové slovo **open** (podobné jako **using** v `C#`), které rozšiřuje scope o daný jmenný prostor nebo modul:
open `System.IO`   nebo **open** `System;` **open** `IO`

Pokud je nějaká funkce definována v několika otevřených prostorech a/nebo v aktuálním modulu, vybere se vždy nejdřív ta z aktuálního modulu a potom z prostoru, který byl otevřen nejpozději. Tedy v případě

```
open List
open Seq
let f () = empty
```

použije `Seq.empty`, i když existuje i `List.empty`.

Je možné použít atribut `[<AutoOpen>]`, který je možné použít:
- ♣ na assembly –– povinný *`string`*ový argument je jméno jmenného prostoru nebo modulu, který je otevřen, je–li assembly referencována při kompilaci
- ♣ na modul –– pak je bez parametrů, a automaticky otevře tento modul, když dojde k otevření jmenného prostoru nebo modulu, ve kterém je obsažen

Při kompilaci jsou automaticky otevřeny následující jmenné prostory (některé obsahují `AutoOpen` moduly):

```
Microsoft.FSharp.Core
Microsoft.FSharp.Core.Operators
Microsoft.FSharp.Collections
Microsoft.FSharp.Control
Microsoft.FSharp.Text
```

Access control

Moduly, typy, metody, hodnoty, funkce, properties a explicit members mají vždy nějaká přístupová práva.
♣ **public** znamená viditelné pro všechny
♣ **internal** znamená viditelné pouze pro tuto assembly
♣ **private** znamená viditelné pouze pro typ nebo modul, ve kterém se nachází
Tato přístupová práva pocházejí od .NETu a mají stejnou sémantiku. Přístup `protected` není možné v `F#`–pu zadefinovat, i když je možné používat typy vytvořené v jiných jazycích, které `protected` mají.

Klíčové slovo **public**, **internal** a **private** se uvádí vždy těsně před jméno definice, takže:

```
let internal f x = x * x
type private tree<'a> = Nil | Node of tree<'a> * tree<'a>
```

Normálně je přístupové právo **public**, kromě **let**ů v definici typů, které jsou vždy **private**.

Není možné upravit přístupová práva jednotlivým konstruktorům algebraických datových typů nebo jednotlivým pojmenovaným položkám recordu.

Signatury aneb přístupová práva na úrovni souboru

Signatury jsou tradiční způsob nastavování přístupových práv, ML ani O'Caml jiný způsob nemá.

Ke každému souboru `.fs` může existovat tzv. signatura. Ta má stejné jméno souboru a koncovku `.fsi`.

Signatura je trochu jako header file. Obsahuje typy všech hodnot/funkcí/typů/modulů zdrojového souboru `.fs`, které budou viditelné i z jiných souborů.

Jednoduchý příklad:

```
sort.fs                                          sort.fsi
namespace MySort                                 namespace MySort

module Helpers =
  let sort xs = ...
  let stable_sort xs = ...

module Sort =                                    module Sort =
  let sort_helper xs = Helpers.sort xs             val sort : 'a list -> 'a list
  let sort xs = sort_helper xs                       when 'a : comparison

module StableSort =                              module StableSort =
  let stable_sort xs = Helpers.stable_sort xs      val stable_sort : 'a list->'a list
                                                     when 'a : comparison
```

Velmi podobného výsledku bychom dosáhli i přidáním slova **private** před `Helpers` a `sort_helper`.

Překladač nám může pomoci s vytvořením souboru `.fsi` –– stačí spustit
`fsc.exe soubor.fs --sig:soubor.fsi` a překladač vytvoří signaturu všech hodnot ve zdrojovém souboru.
Při kompilaci musí být soubor `.fsi` na příkazové řádce před svým zdrojovým souborem `.fs`.

Parsování pomocí fslex a fsyacc

Nejprve si vytvořme datový typ, který budeme chtít načítat:

```
namespace Ast
type Expr = Value of float
          | Plus  of Expr * Expr
          | Minus of Expr * Expr
          | Times  of Expr * Expr
          | Divide of Expr * Expr
```

Pro implementaci načítání potřebujeme dvě věci, tokenizér a parser.

Tokenizér

Tokenizér, někdy také lexer, je proces, který zpracovává vstupní text (buď posloupnosti *byte* nebo *char*) a vykusuje z ní "tokeny". Tyto tokeny mohou být například jednotlivá písmena, ale častěji to jsou operátory, identifikátory, proměnné, klíčová slova atd.

V našem případě budou tokeny
- `NUM` **of** *float* pro číslo
- `LPAREN RPAREN` pro levou a pravou závorku
- `PLUS MINUS ASTER SLASH` pro operátory + – * /

Většinou se zahazují nepodstatné informace jako mezery, nové řádky, komentáře atd.
Tokeny se definují na straně parseru, takže náš tokenizér neobsahuje jejich definici.

Tokenizér se ukládá do souboru `.fsl` a překládá se do `.fs` pomocí
`fslex.exe soubor.fsl [--unicode | --codepage <int>]`
Tokenizér vypadá jako

```
{ header }
let ident = regexp
let ident = regexp
...
rule entrypoint [arg1... argn] =
  parse regexp { action }
      | ...
      | regexp { action }
and entrypoint [arg1... argn] =
  parse ...
and ...
{ trailer }
```

Regexpy vypadají jako

- `'znak'`
- `"řetězec"`
- `[znaky]`
- `[^znaky]`
- `regexp*`
- `regexp+`
- `regexp?`
- `regex1 | regex2`
- `regexp1 regexp2` konkatenace, nejprve první a pak druhý
- `ident` namatchuje to dříve zadefinovaný regexp
- `eof` konec vstupu

Akce je libovolný výraz, může používat proměnnou `lexbuf`, která je typu `LexBuffer<byte>` nebo `LexBuffer<char>`, podle toho, jestli je to 8–bitovy nebo unicode lexer, a ve které je namatchovaný aktuální regexp na levé straně. `LexBuffer<'a>` nabizi:

```
member lexBuffer.Lexeme : 'char array
member lexBuffer.LexemeChar (arg1 : int) -> 'char
member lexBuffer.LexemeLength : int
member lexBuffer.StartPos : Position
member lexBuffer.EndPos : Position
member lexBuffer.IsPastEndOfStream : bool
static member LB.FromBytes (arg1 : byte []) : LexBuffer<byte>
static member LB.FromChars (arg1 : char []) : LexBuffer<char>
static member LB.FromString (arg1 : string) : LexBuffer<char>
static member LB.FromBinaryReader (arg1:System.IO.BinaryReader):LexBuffer<byte>
static member LB.FromTextReader (arg1 : System.IO.TextReader) : LexBuffer<char>
static member LB.FromFunction (arg1 : ('char []*int*int->int) : LexBuffer<'char>
static member LB.FromAsyncFunction (arg1 : ('char []*int*int->Async<int>)
                                     : LexBuffer<'char>
static member LB.LexemeString (arg1 : LexBuffer<char>) : string
```

Ukázkový lexer

```
{
module Lexer
open Parser; open Microsoft.FSharp.Text.Lexing
let lexeme lexbuf = LexBuffer<char>.LexemeString lexbuf
}
let digit = ['0'-'9']
let whitespace = [' ' '\t' ]
let newline = ('\n' | '\r' '\n')
rule tokenize = parse
| whitespace    { tokenize lexbuf }
| newline       { lexbuf.EndPos <- lexbuf.EndPos.NextLine; tokenize lexbuf }
| "+"           { PLUS }   | "-"          { MINUS }   | "*"          { ASTER }
| "/"           { SLASH }  | "("          { LPAREN }  | ")"          { RPAREN }
| ['-']?digit+('.'digit+)?(['e''E']digit+)?  { NUM <| float (lexeme lexbuf) }
| eof   { EOF }
```

Tokenizéry mohou mít libovolný počet parametrů, se kterými je musíme volat. Pokud chceme, aby byly perzistentní, můžeme použít následující trik:

```
rule tokenize_step a b = ...
{
let tokenize = let a, b = ref a_init, ref b_init
               fun lexbuf -> tokenize_step a b lexbuf
}
```

Parser

Parser je LALR(1) parser, jako `ocamlyacc` nebo `bison`. Ukládá se do souboru `.fsy` a překládá se do `.fs` pomocí
`fsyacc.exe soubor.fsy --module jmeno_vysledneho_modulu [--codepage <int>]`
Parser vypadá jako

```
%{
  header
%}
  declarations
%%
  rules
%%
  trailer
```

Části `header` a `trailer` jsou libovolný F# kód. `Declarations` definuje hlavní pravidlo gramatiky pomocí `%start jméno_tokenu` a také jeho typ pomocí `%type <typ> jméno_tokenu`. Dále definuje jednotlivé tokeny pomocí `%token jméno1 jméno2 ...` pro tokeny bez dat a pomocí `%token <typ> jméno1 jméno2` … pro tokeny s daty.

`Rules` definuje pravidla gramatiky jako

```
pravidlo :
   | token_ci_pravidlo1 token_ci_pravidlo2 … { akce, $i je i-tý token/pravidlo }
   | token_ci_pravidlo1 token_ci_pravidlo2 … { akce, $i je i-tý token/pravidlo }
  …
```

Příklad k naší kalkulačce

```
%{
open Ast
%}
%start start
%type < Ast.Expr > start
%token <float> NUM
%token PLUS MINUS ASTER SLASH LPAREN RPAREN EOF
%%
start: Expr EOF                         { $1 }
Expr: Expr PLUS  Term                   { Plus ($1, $3)   }
    | Expr MINUS Term                   { Minus ($1, $3)  }
    | Term                              { $1              }
Term: Term ASTER Factor                 { Times ($1, $3)  }
    | Term SLASH Factor                 { Divide ($1, $3) }
    | Factor                            { $1              }
Factor: NUM                             { Value $1        }
      | LPAREN Expr RPAREN              { $2              }
```

Gramatika umí nastavovat jednotlivým pravidlům a tokenům priority a asociativitu, takže náš příklad můžeme vylepšit:

```
%token <float> NUM
%token PLUS MINUS ASTER SLASH LPAREN RPAREN EOF
%left PLUS MINUS        /* nejnižší */
%left ASTER SLASH       /* střední  */
%nonassoc UMINUS        /* nejvyšší */
%%
start: Expr EOF                      { $1 }
Expr: NUM                            { Value $1             }
    | LPAREN Expr RPAREN             { $2                   }
    | Expr PLUS  Expr                { Plus ($1, $3)        }
    | Expr MINUS Expr                { Minus ($1, $3)       }
    | Expr ASTER Expr                { Times ($1, $3)       }
    | Expr SLASH Expr                { Divide ($1, $3)      }
    | MINUS Expr %prec UMINUS        { Minus (Value 0., $2) }
```

Tři slova `%left`, `%right` a `%nonassoc` určují asociativitu zmíněných tokenů.
Zároveň určují i prioritu –– každá řádka je jedna priorita, řádky jsou v pořadí od nejnižší po tu s nejvyšší prioritou.

Každé pravidlo má normálně prioritu stejnou jako jeho nejpravější token. Je možné ji upravit pomocí `%prec token`, které se uvádí na konec pravidla –– to má potom prioritu stejnou, jako uvedený token.

V parseru vznikne metoda `start: (LexBuffer<'cty>->token) -> LexBuffer<'cty> -> Ast.Expr`

```
try Parser.start (Lexer.tokenize) lexbuf
with e -> let pos = lexbuf.StartPos
          failwithf "Error at line %d col %d: %s" pos.Line pos.Column e.Message
```