BPRD – Programs as Data

Lecture 3: FsLexYacc

Niels Hallenberg

GitHub:
`http://fsprojects.github.io/FsLexYacc/index.html`

`Expr` as running example

File `Absyn.fs` defines the type representing the abstract syntax tree that the parser builts.

```
module Absyn

type expr =
  | CstI of int
  | Var of string
  | Let of string * expr * expr
  | Prim of string * expr * expr
```

Expr as running example

File ExprLex.fsl contains the lexer specification.

```
...
rule Token = parse
  | [' ' '\t' '\r'] { Token lexbuf }
  | '\n'            { lexbuf.EndPos <- lexbuf.EndPos.NextLine; Token l
  | ['0'-'9']+      { CSTINT (System.Int32.Parse (lexemeAsString lexbu
  | ['a'-'z''A'-'Z']['a'-'z''A'-'Z''0'-'9']*
                    { keyword (lexemeAsString lexbuf) }
  | '+'             { PLUS  }
  | '-'             { MINUS }
  | '*'             { TIMES }
...
```

The tokens defined in the parser specification, ExprPar.fsy, is used in the lexer specification.

`Expr` as running example

File `ExprPar.fsy` contains the parser specification including definitions of tokens and the inclusion of the type representing the abstract syntax tree.

```
...
%token <int> CSTINT
%token <string> NAME
%token PLUS MINUS TIMES DIVIDE EQ
%token END IN LET
%token LPAR RPAR
%token EOF
...

Main:
    Expr EOF                            { $1                }
;

Expr:
    NAME                                { Var $1            }
  | CSTINT                              { CstI $1           }
  | MINUS CSTINT                        { CstI (- $2)       }
  | LPAR Expr RPAR                      { $2                }
  | LET NAME EQ Expr IN Expr END        { Let($2, $4, $6)   }
...
```

The `FsLexYacc` package

An installation of `FsLexYacc` contains the following files

- A runtime dll to include in your project, e.g., `FsLexYacc.Runtime.dll`
- The lexer generator executable `fslex.dll`
- The parser generator executable `fsyacc.dll`

And then some supporting files coming with the package.

To setup FsLexYacc you basically need to make sure these files are accessible.

Invoking fsyacc

The first task is to use the parser generator on the parser specification.

```
$ fsyacc --module ExprPar ExprPar.fsy
```

This will build a file `ExprPar.fs` containing the generated F# code representing the parser:

```
$ fsyacc --module ExprPar ExprPar.fsy
building tables
computing first function...time: 00:00:00.0713506
building kernels...time: 00:00:00.0371238
...
$ ll ExprPar.fs
-rw-r--r-- 1 nh staff 12619 Oct 12 21:56 ExprPar.fs
```

Invoking fsyacc

Investigating the file `ExprPar.fs` we find a datatype for the tokens defined and used by the lexer generator.

```
// Implementation file for parser generated by fsyacc
module ExprPar
...
// This type is the type of tokens accepted by the parser
type token =
  | EOF
  | LPAR
  | RPAR
  | END
  | IN
  | LET
  | PLUS
  | MINUS
  | TIMES
  | DIVIDE
  | EQ
  | NAME of (string)
  | CSTINT of (int)
```

Invoking fslex

The second task is to use the lexer generator on the lexer specification.

```
$ fslex --unicode ExprLex.fsl
```

This will build a file `ExprLex.fs` containing the generated F# code representing the lexer. The lexer depends on the module `ExprPar.fs` containing the type for the tokens.

```
$ fslex --unicode ExprLex.fsl
compiling to dfas (can take a while...)
15 states
writing output
$ ll ExprLex.fs
-rw-r--r-- 1 nh staff 23452 Oct 12 22:04 ExprLex.fs
```

Invoking fslex and fsyacc

- Build the lexer and parser vs files `ExprLex.fs` and `ExprPar.fs`
- Compile as modules together with `Absyn.fs` and `Parse.fs`:

```
$ fsyacc --module ExprPar ExprPar.fsy
$ fslex --unicode ExprLex.fsl
$ dotnet fsi -r ~/fsharp/FsLexYacc.Runtime.dll
                Absyn.fs ExprPar.fs ExprLex.fs Parse.fs
```

- Open the Parse module and experiment:

```
open Parse;;
fromString "x + 52 * wk";;
```

Lexing - get one token at the time (`fsLexYacc.fsx`)

From string to lexbuffer to tokens.

```
let str = "2+4"
let lexbuf =  Lexing.LexBuffer<char>.FromString(str)

let nextToken (lexbuf : Lexing.LexBuffer<char>) : ExprPar.toke
  if not lexbuf.IsPastEndOfStream
  then ExprLex.Token lexbuf
  else ExprPar.EOF
```

Execute `nextToken` until you get `EOF`:

```
nextToken lexbuf;;
val it : ExprPar.token = CSTINT 2
> nextToken lexbuf;;
val it : ExprPar.token = PLUS
> nextToken lexbuf;;
val it : ExprPar.token = CSTINT 4
> nextToken lexbuf;;
val it : ExprPar.token = EOF
```

Lexing - get a list of tokens (`fsLexYacc.fsx`)

From string to lexbuffer to list of tokens.

```
let str = "2+4"
let lexbuf =  Lexing.LexBuffer<char>.FromString(str)

let listTokens (lexbuf : Lexing.LexBuffer<char>) =
  let rec listTokens’ () =
    if lexbuf.IsPastEndOfStream
    then []
    else ExprLex.Token lexbuf :: listTokens’ ()
  listTokens’ ()
```

Execute `listTokens` to get the list of tokens:

```
> let lexbuf =  Lexing.LexBuffer<char>.FromString(str);;
val lexbuf: Lexing.LexBuffer<char>

> listTokens lexbuf;;
val it : ExprPar.token list = [CSTINT 2; PLUS; CSTINT 4; EOF]
```

Putting together lexer and parser (`fsLexYacc.fsx`)

From lexbuffer to tokens to abstract syntax tree.

```
let parse (lexbuf : Lexing.LexBuffer<char>) =
  try
    ExprPar.Main ExprLex.Token lexbuf
  with
    | exn -> let pos = lexbuf.EndPos
             failwithf "..."
                (exn.Message) (pos.Line+1) pos.Column
```

Execute `parse` to get the abstract syntax tree:

```
> let lexbuf =  Lexing.LexBuffer<char>.FromString(str);;
val lexbuf: Lexing.LexBuffer<char>

> parse lexbuf;;
val it : expr = Prim ("+",CstI 2,CstI 4)
```

Setup

Please visit

- Course repo under `fsharp`.
- This course will make use of dotnet tool, such that Mono is not necessary on Linux and Mac.
- `http://fsprojects.github.io/FsLexYacc/`