

FsYacc: Extra Web Material

The following is taken from the F# Wiki, written by one of the authors.

fsyacc.exe is a [LALR](#) parser generator. It follows essentially the same specification as the [OCamlYacc](#) parser generator, especially when used with the 'ml compatibility' switch.

Sample input

Parser generators typically produce [Abstract Syntax Trees](#) represented by values in an F# [Algebraic Data Type](#). For example, the following is taken (with permission) from the F# sample samples\fssharp\Parsing:

```
type expr =
```

```
| Val of string
```

```
| Int of int
```

```
| Float of float
```

```
| Decr of expr
```

```
type stmt =
```

```
| Assign of string * expr
```

```
| While of expr * stmt
```

```
| Seq of stmt list
```

```
| IfThen of expr * stmt
```

```
| IfThenElse of expr * stmt * stmt
```

```
| Print of expr
```

```
type prog = Prog of stmt list
```

Given that, a typical parser specification is as follows:

```
%{
```

open Ast

%}

%start start

%token <string> ID

%token <System.Int32> INT

%token <System.Double> FLOAT

%token DECR LPAREN RPAREN WHILE DO END BEGIN IF THEN ELSE PRINT SEMI ASSIGN EOF

%type < Ast.prog > start

%%

start: Prog { \$1 }

Prog: StmtList { Prog(List.rev(\$1)) }

Expr: ID { Val(\$1) }

| INT { Int(\$1) }

| FLOAT { Float(\$1) }

| DECR LPAREN Expr RPAREN { Decr(\$3) }

```

Stmt: ID ASSIGN Expr { Assign($1,$3) }
  | WHILE Expr DO Stmt { While($2,$4) }
  | BEGIN StmtList END { Seq(List.rev($2)) }
  | IF Expr THEN Stmt { IfThen($2,$4) }
  | IF Expr THEN Stmt ELSE Stmt { IfThenElse($2,$4,$6) }
  | PRINT Expr { Print($2) }

```

```

StmtList: Stmt { [$1] }
  | StmtList SEMI Stmt { $3 :: $1 }

```

The above generates a datatype for tokens and a function for each 'start' production. Parsers are typically combined with a lexer generated using [fslex](#). For example, a program AST can be generated from a file using:

```

let parse() =
  let stream = new StreamReader("myfile.txt") in
  let myProg =
    let lexbuf = Lexing.from_stream_reader stream in
    Pars.start Lex.token lexbuf in
  myProg

```

Command line options

```
fsyacc <filename>
```

```
-o Name the output file.
```

```
-v Produce a listing file.
```

```
--module Define the F# module name to host the generated parser.
```

```
--open Add the given module to the list of those to open in both the generated signature and implementation.
```

```
--ml-compatibility Support the use of the global state from the 'Parsing' module in MLLib.
```

```
--tokens Simply tokenize the specification file itself.
```

```
--help Display this list of options
```

Managing and using position markers

Each action in an fsyacc parser has access to a parseState value through which you can access position information.

```
type IParseState<'pos> =  
  
  interface  
  
    abstract StartOfRHS: int -> 'pos  
  
    abstract EndOfRHS : int -> 'pos  
  
    abstract StartOfLHS: 'pos  
  
    abstract EndOfLHS : 'pos  
  
    abstract GetData  : int -> obj  
  
    abstract RaiseError<'b> : unit -> 'b  
  
  end
```

These are fairly self explanatory - RHS relate to the indexes of the items on the right hand side of the current production, the LHS relates to the entire range covered by your production. You shouldn't use "GetData" directly - these is called automatically by \$1, \$2 etc. You can call [RaiseError](#) if you like.

The 'pos position values carried by the lexer and parser can in theory be any type you wish. However in practice people always use the position values defined in the "Lexing" module, i.e. see

http://research.microsoft.com/fsharp/manual/mlib/Microsoft.FSharp.Compatibility.OCaml.Lexing.type__position.html

These have the following somewhat obscure definition (blame [OCaml](#) compatibility for how cryptic this is!)

```
type position =  
  
  {  
  
    /// Absolute offset of position. Automatically calculated  
  
    /// by lexer  
  
    pos_cnum : int;  
  
  
  
    /// Filename, based name given in initial position
```

```

pos_fname : string;

/// Line number - must be updated by user lexer actions
/// Must be updated by user lexer actions.
pos_lnum : int;
/// Absolute offset of beginning of line.
/// Must be updated by user lexer actions.
pos_bol : int;
}

```

You must set the initial position when you create the lexbuf:

```

let setInitialPos (lexbuf:lexbuf) filename =
  lexbuf.EndPos <-{ pos_bol = 0;
                  pos_fname=filename;
                  pos_cnum=0;
                  pos_lnum=1 }

```

You must also update the position recorded in the lex buffer each time you process what you consider to be a new line:

```

let newline (lexbuf:lexbuf) =
  lexbuf.EndPos <- lexbuf.EndPos.AsNewLinePos()

```

Likewise if your language includes the ability to mark source code locations (e.g. the #line directive in [OCaml](#) and F#) then you must similarly adjust the lexbuf.EndPos according to the information you grok from your input.

Notes on [OCaml](#) Comaptibility

Xuhui Li pointed out that [OCaml](#) Yacc accepts the following:

```
%type < context -> context > toplevel
```

For [FsYacc](#) you just add parentheses:

```
%type < (context -> context) > toplevel
```