Parsing F# with PetitParser

Milan Kubicek
SCG Seminar
29.09.2015

F# specification

expr ← expr expr

expr op expr

ident

[op] const

op ← '+'

300 pages

230 production rules

context-free form

Left recursion

F# spec

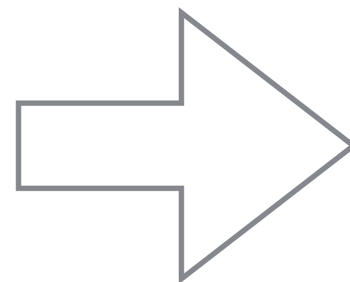
expr ← expr expr

expr op expr

ident

[op] const

op ← ‘+’

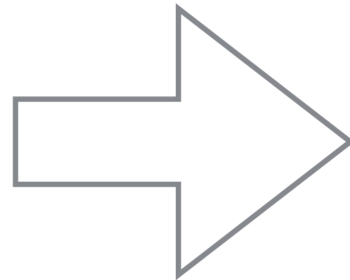


“It’s dead Jim”

Left recursion in grammars cause VM crash!

Eliminating left recursion

```
expr ← expr expr
       expr op expr
       ident
       [op] const

  op ← '+'
```

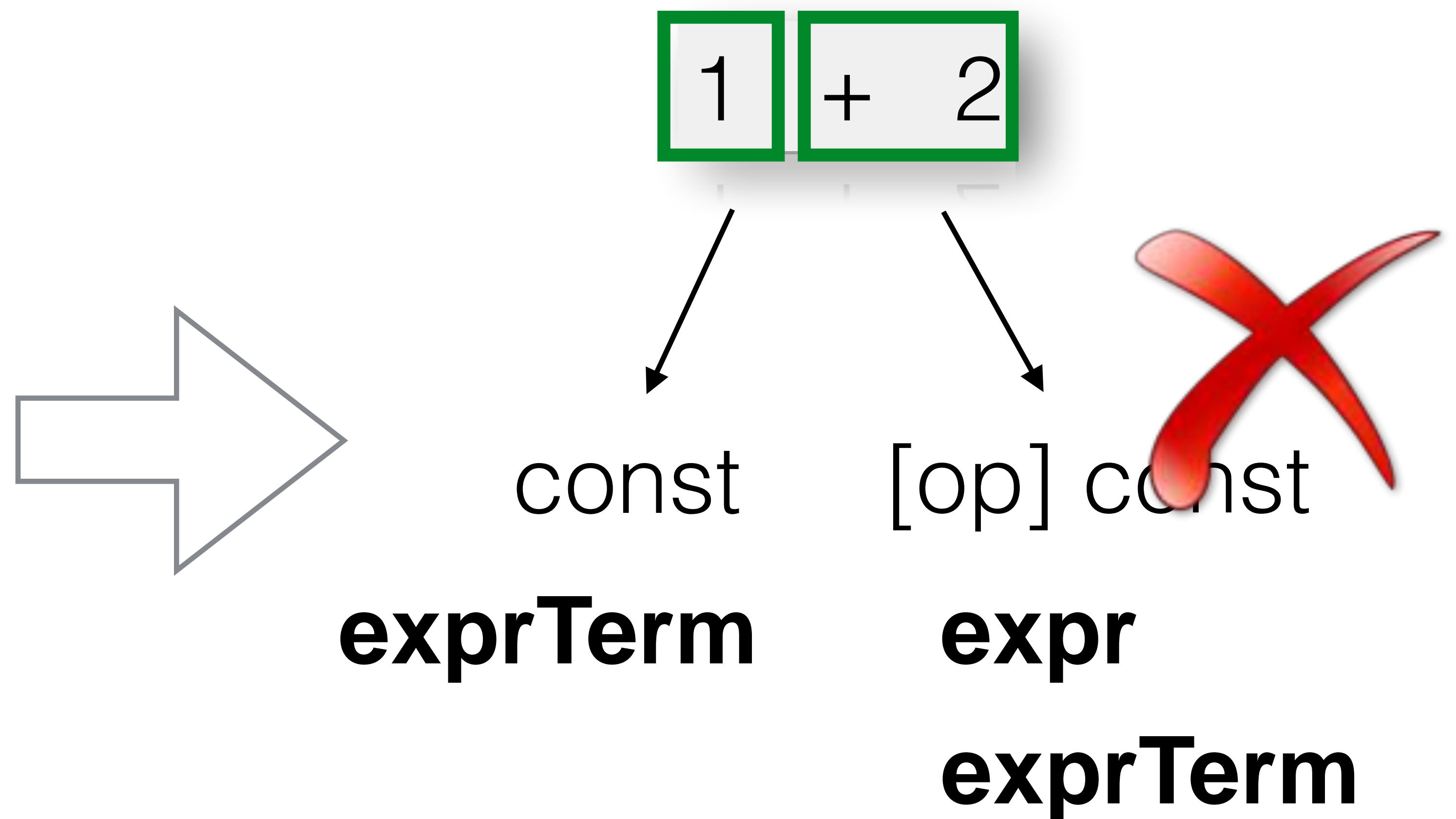


```
    expr ← exprTerm expr
           exprTerm op expr
           exprTerm

exprTerm ← ident
           [op] const

      op ← '+'
```

In the right-hand grammar, the word exprTerm is printed in bold.

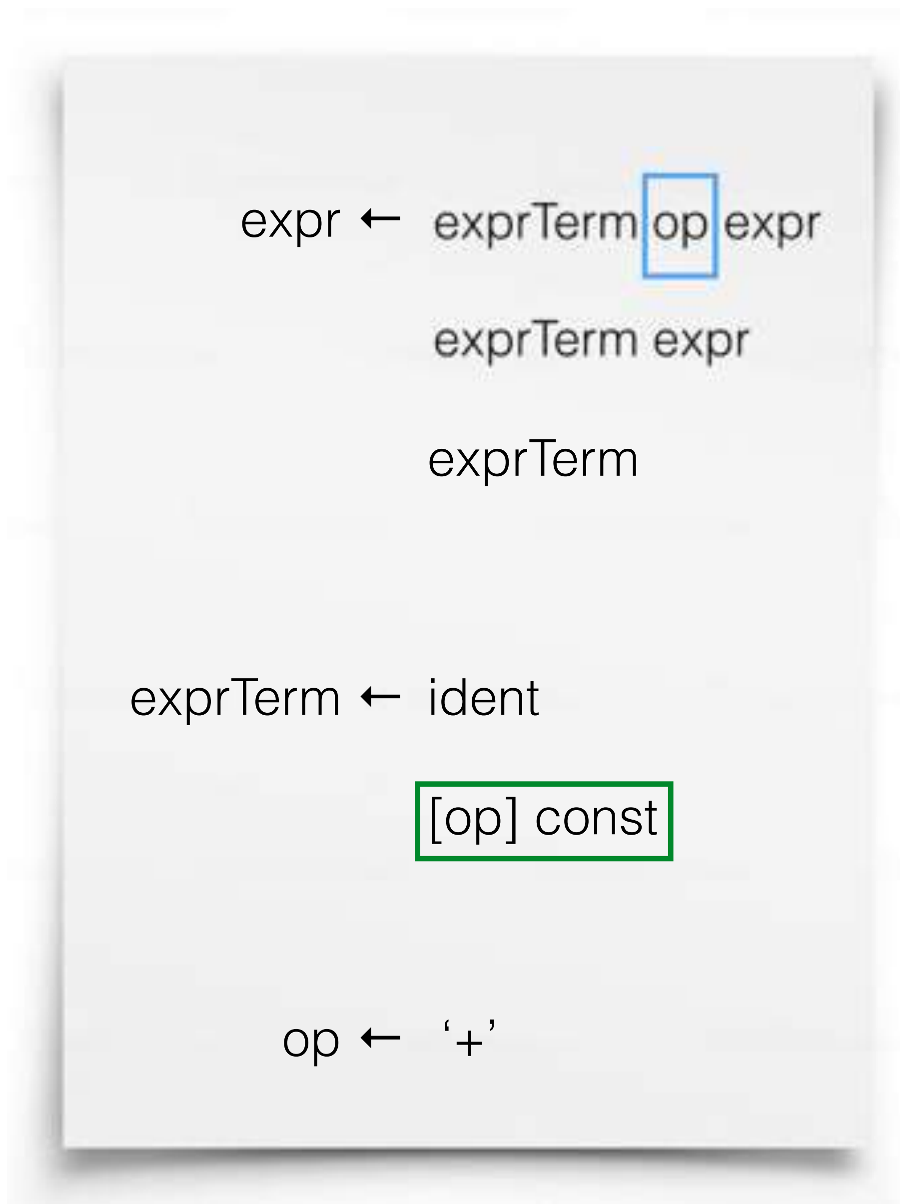
Ordered choice in PEG

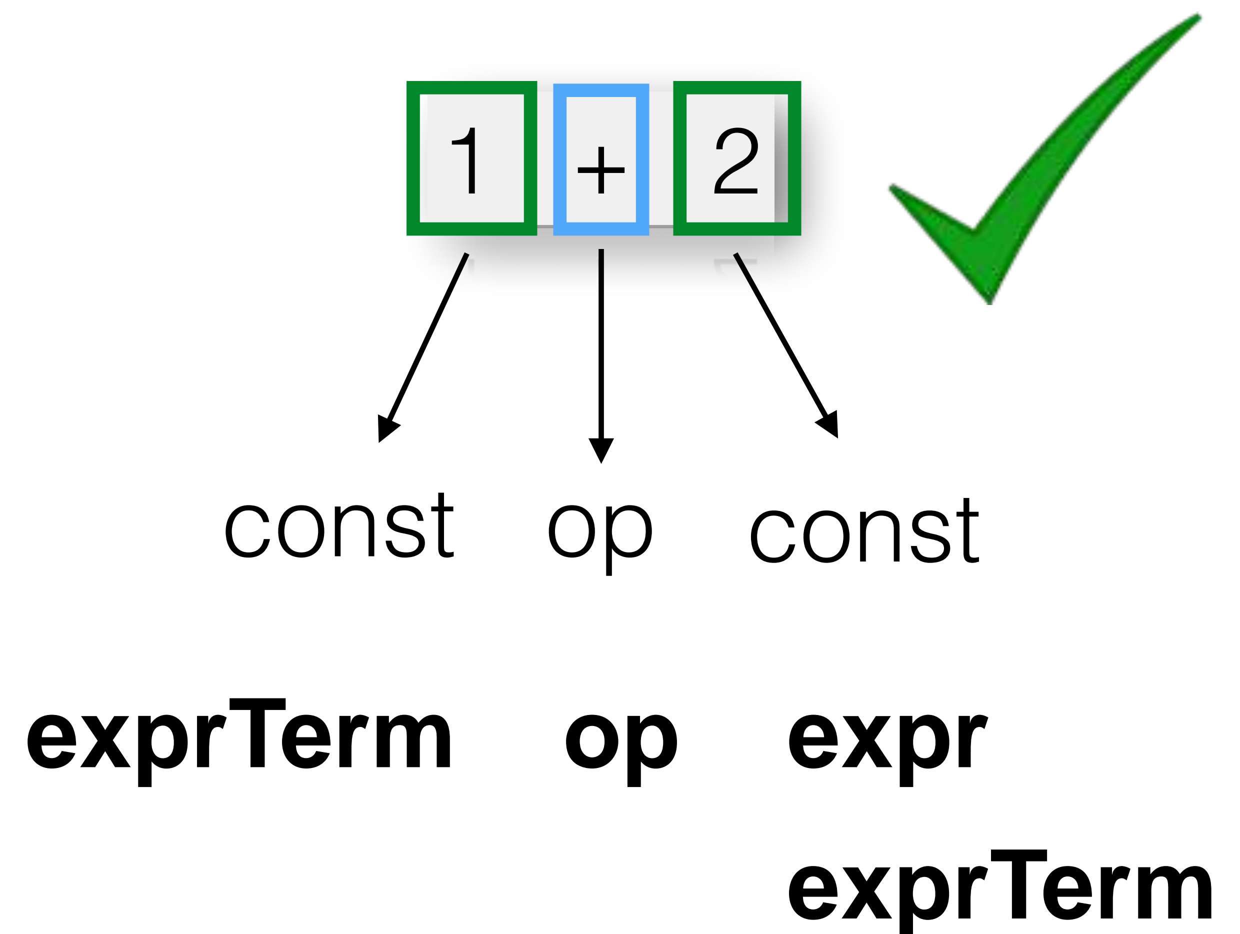
```
expr ← exprTerm expr
       exprTerm op expr
       exprTerm

exprTerm ← ident
           [op] const

op ← '+'
```



If the first alternative succeeds, the second alternative is ignored.

Ordered choice in PEG





Performance

expr ← exprTerm op expr

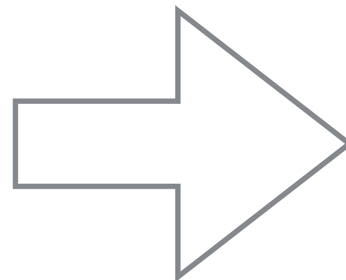
exprTerm expr

exprTerm

exprTerm ← ident

[op] const

op ← '+'



1

- packrat parsing not enabled by default
- two possibilites:
  - refactor grammar
  - enable packrat (parser memoization)

Grammar transformation

```
expr ← expr expr
       expr op expr
       ident
       [op] const

  op ← '+'
```

```
    expr ← exprTerm op expr
           exprTerm expr
           exprTerm

exprTerm ← ident
           [op] const

      op ← '+'
```

..in the wild (1)

```
expr :
      const
      ( expr )
      begin expr end
      long-ident-or-op
      expr '.' long-ident-or-op
      expr expr
      expr(expr)
      expr<types>
      expr infix-op expr
      prefix-op expr
      expr.[expr]
      expr.[slice-range]
      expr.[slice-range, slice-range]
      expr <- expr
      expr , ... , expr
      new type expr
      { new base-call object-members interface-impls }
      { field-initializers }
      { expr with field-initializers }
      [ expr ; ... ; expr ]
      [| expr ; ... ; expr |]
      expr { comp-or-range-expr }
      [ comp-or-range-expr]
      [| comp-or-range-expr |]
      lazy expr
      null
      expr : type
      expr :> type
      expr :? type
      expr :?> type
      upcast expr
      downcast expr
      let function-defn in expr
      let value-defn in expr
      let rec function-or-value-defns in expr
      use ident = expr in expr
      ...
```

```
...
expr '.' long-ident-or-op
expr expr
expr(expr)
expr<types>
expr infix-op expr
prefix-op expr
expr.[expr]
expr.[slice-range]
expr.[slice-range, slice-range]
expr <- expr
expr , ... , expr
new type expr
{ new base-call object-members interface-impls }
{ field-initializers }
{ expr with field-initializers }
[ expr ; ... ; expr ]
[| expr ; ... ; expr |]
expr { comp-or-range-expr }
[ comp-or-range-expr]
[| comp-or-range-expr |]
lazy expr
null
expr : type
expr :> type
expr :? type
expr :?> type
upcast expr
downcast expr
```

p. 272, F# 3.0 language specification

..in the wild (2)

<table>
<tr><th>Operator or expression</th><th>Associativity</th></tr>
<tr><td>f<types></td><td>Left</td></tr>
<tr><td>f(x)</td><td>Left</td></tr>
<tr><td>.</td><td>Left</td></tr>
<tr><td>prefix-op</td><td>Left</td></tr>
<tr><td>"| rule"</td><td>Right</td></tr>
<tr><td>"f x"
"lazy x"
"assert x"</td><td>Left</td></tr>
<tr><td>**OP</td><td>Right</td></tr>
<tr><td>*OP /OP %OP</td><td>Left</td></tr>
<tr><td>-OP +OP</td><td>Left</td></tr>
<tr><td>:?</td><td>Not associative</td></tr>
<tr><td>::</td><td>Right</td></tr>
<tr><td>^OP</td><td>Right</td></tr>
<tr><td>!=OP <OP >OP = |OP &OP $</td><td>Left</td></tr>
<tr><td>:> :?></td><td>Right</td></tr>
</table>

...

Rules of precedence, p. 35, F# 3.0 language specification

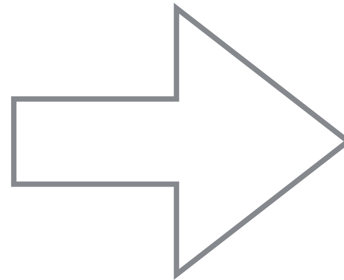
..in the wild (3)

```
...
additional-constr-defn :
     attributes_opt access_opt new pat as-defn = additional-constr-expr

additional-constr-expr :
!    stmt ';' additional-constr-expr
     additional-constr-expr then expr
     if expr then additional-constr-expr else additional-constr-expr
 !   let val-decls in  additional-constr-expr
     additional-constr-init-expr

additional-constr-init-expr :
     '{' class-inherits-decl field-initializers '}'
!    new type expr
...
```

p. 279, F# 3.0 language specification

Results (1): tool for left recursion detection

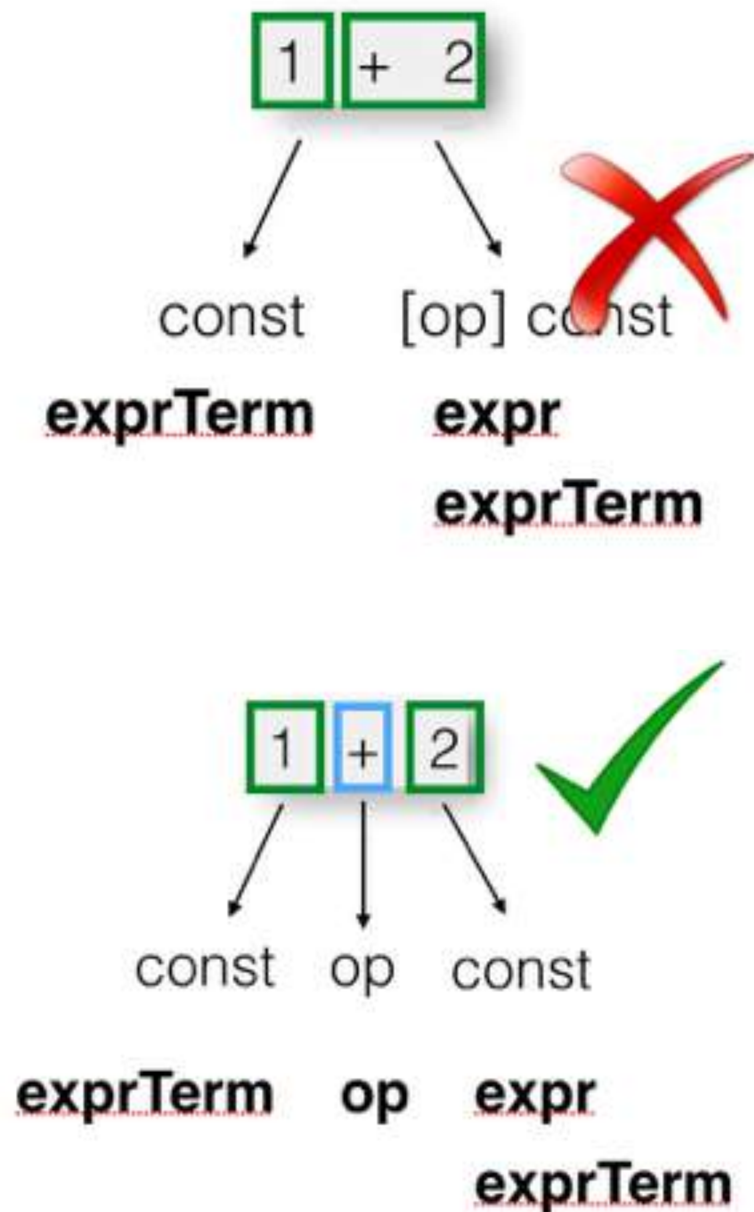
“It’s dead Jim”

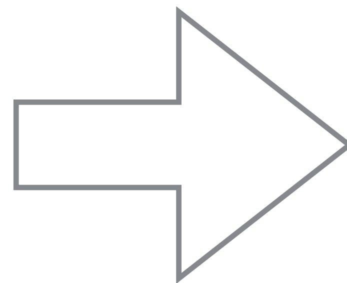


```
petitGrammar hasLeftRecursion
    ifTrue: [ petitGrammar getLeftRecursion  ].
```

“offline” and “online” algorithm for left recursion detection in PetitParser grammars

Results (2): AST extractor for validation





F# compiler services
AST extractor

Generates AST for validation

Results (3): transformed, validated F# syntax rules

- implemented all expression rules and in total 170 out of 230 specified rules
- validated them with 8000 LOC verbose F# code

Conclusions

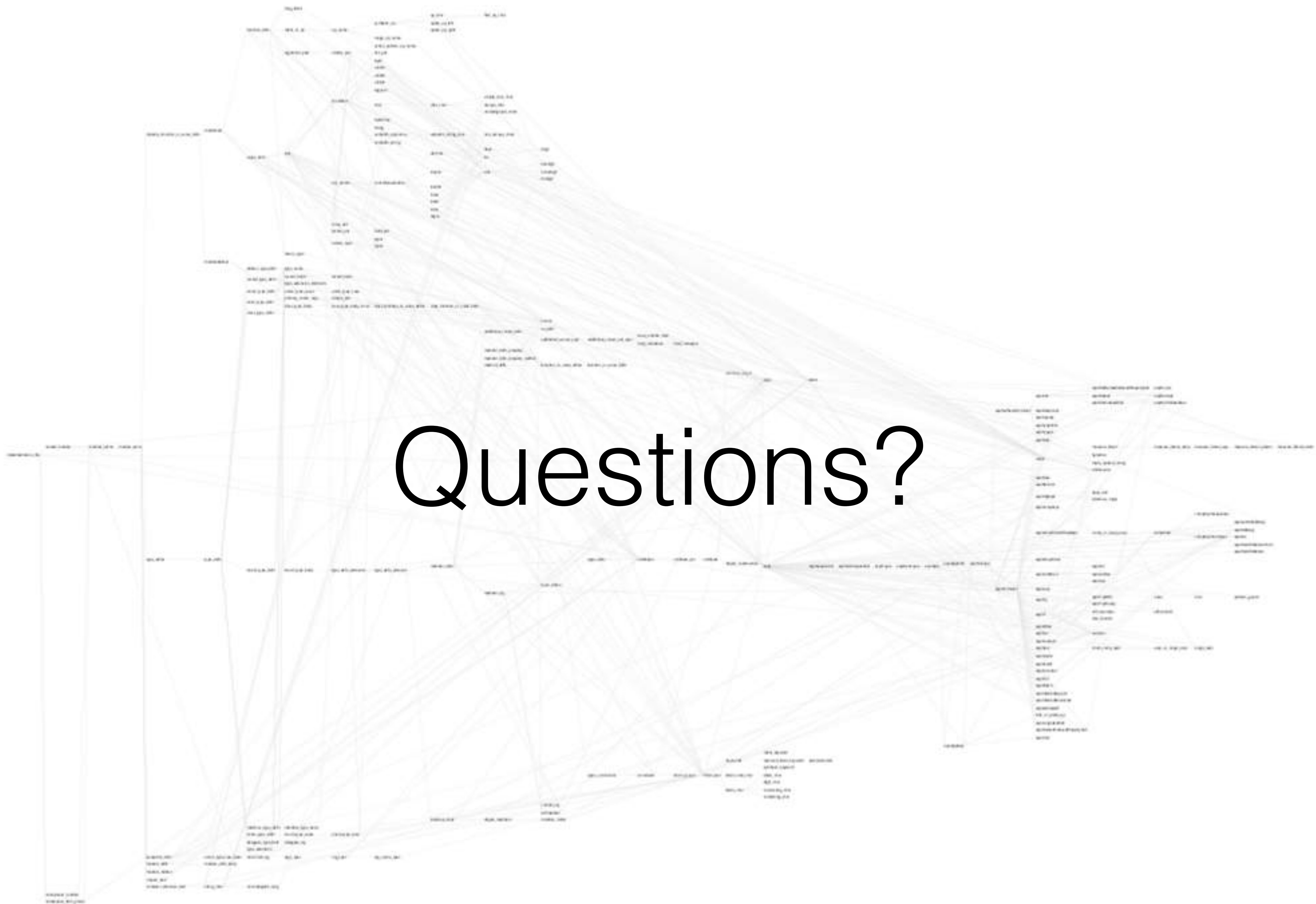
The F# CFG to PEG conversion requires quite some manpower for:

a) left recursion elimination
→ left recursion detection tool

b) ordered choice / priority modification
→ F# AST extractor for validation

c) performance issues

d) slight differences between F# implementation and specification


Questions?

F#

- open source, cross-platform compiler
- 3.1 stable release
- 3.0 specification final
- originated from ML and has been influenced by OCaml, C#, Python, Haskell, Scala and Erlang

verbose syntax

```
#light "off"

let rec fibonacci n =
    if (n = 1 || n = 2) then
        1
    else
        let result = fibonacci(n-1)
                   + fibonacci(n-2) in
        result
in

for i in 1 .. 10 do
    printfn "%d: %d" i (fibonacci i)
done
```

lightweight syntax

```
#light "on"

let rec fibonacci n =
    if (n = 1 || n = 2) then
        1
    else
        let result = fibonacci(n-1)
                   + fibonacci(n-2)
        result

for i in 1 .. 10 do
    printfn "%d: %d" i (fibonacci i)
```

eliminating left-recursion

1. *For each nonterminal* A_i*:*
    1. *Repeat until an iteration leaves the grammar unchanged:*
        1. *For each rule* $A_i \rightarrow \alpha_i$, α_i *being a sequence of terminals and nonterminals:*
            1. *If* α_i *begins with a nonterminal* A_j *and* $j < i$*:*
                1. *Let* β_i *be* α_i *without its leading* A_j.
                2. *Remove the rule* $A_i \rightarrow \alpha_i$.
                3. *For each rule* $A_j \rightarrow \alpha_j$*:*
                    1. *Add the rule* $A_i \rightarrow \alpha_j \beta_i$.
    2. *Remove direct left recursion for* A_i *as described above.*