

# BNFC-meta — an Embedded Parser Generator

Jonas Duregård   Patrik Jansson

Chalmers University of Technology

September 22, 2011

Open positions: 2 Assistant Professors in Functional Programming  
Deadline: 2011-10-18. [RAWFP = Resource Aware FP project]

## Example

*wait* :: *Expr* → *Stm*

*wait* *steps* = [*stm* | *while* (*a* < %*steps*) { *a* = *a* + 1; } ||]

# Embedded or standalone DSL?

(with examples from the grammar/parsing domain)

- Embedded DSL (Parser combinators)
  - The language is implemented as a *library*
  - Parsers are *combined* directly from library functions
  - Some advantages: Familiarity, Abstraction
  
- Standalone DSL (Parser generator)
  - The language is implemented in a *utility program*
  - Parser source code is *generated* from a grammar file
  - Some advantages: Static analysis, Streamlined syntax

# Embedded or standalone DSL?

(with examples from the grammar/parsing domain)

- Embedded DSL (Parser combinators)
  - The language is implemented as a *library*
  - Parsers are *combined* directly from library functions
  - Some advantages: Familiarity, Abstraction
- Semi-embedded DSL (BNFC-meta)
  - The language is implemented as a *library*
  - Parser source code is *generated* directly by the library functions
  - Sometimes called a *two-staged* DSL, or an *embedded compiler*.
- Standalone DSL (Parser generator)
  - The language is implemented in a *utility program*
  - Parser source code is *generated* from a grammar file
  - Some advantages: Static analysis, Streamlined syntax

# An example

A tiny DSL for regular expressions:

```
data Regex = C Char | Rep Regex | Regex :>: Regex
```

and an example we may want to implement:

```
isAsB :: String → Bool  -- does the string match a* b ?
```

# Building block 1: Template Haskell

Template Haskell [Sheard & Peyton-Jones Haskell'02] blurs the edges between embedded and semi-embedded

Embedded DSL (deep embedding):

$$\begin{aligned} \text{runReg} &:: \text{Regex} \rightarrow \text{String} \rightarrow \text{Bool} \\ \text{isAsB} &= \text{runReg} (\text{Rep} (\text{C 'a'}) \text{:>:} \text{C 'b'}) \end{aligned}$$

# Building block 1: Template Haskell

Template Haskell [Sheard & Peyton-Jones Haskell'02] blurs the edges between embedded and semi-embedded

Embedded DSL (deep embedding):

$$\begin{aligned} \text{runReg} &:: \text{Regex} \rightarrow \text{String} \rightarrow \text{Bool} \\ \text{isAsB} &= \text{runReg} (\text{Rep} (\text{C 'a'}) \text{:>:} \text{C 'b'}) \end{aligned}$$

Semi-embedded DSL (with  $\text{compileReg} :: \text{Regex} \rightarrow Q \text{Exp}$ ):

$$\text{isAsB} = \$ (\text{compileReg} (\text{Rep} (\text{C 'a'}) \text{:>:} \text{C 'b'}))$$

# Building block 1: Template Haskell

Template Haskell [Sheard & Peyton-Jones Haskell'02] blurs the edges between embedded and semi-embedded

Embedded DSL (deep embedding):

$$\begin{aligned} \text{runReg} &:: \text{Regex} \rightarrow \text{String} \rightarrow \text{Bool} \\ \text{isAsB} &= \text{runReg} (\text{Rep} (\text{C 'a'}) \text{:>:} \text{C 'b'}) \end{aligned}$$

Semi-embedded DSL (with  $\text{compileReg} :: \text{Regex} \rightarrow Q \text{Exp}$ ):

$$\text{isAsB} = \$ (\text{compileReg} (\text{Rep} (\text{C 'a'}) \text{:>:} \text{C 'b'}))$$

# Building block 1: Template Haskell

Template Haskell [Sheard & Peyton-Jones Haskell'02] blurs the edges between embedded and semi-embedded

Embedded DSL (deep embedding):

$$\begin{aligned} \text{runReg} &:: \text{Regex} \rightarrow \text{String} \rightarrow \text{Bool} \\ \text{isAsB} &= \text{runReg} (\text{Rep} (\text{C 'a'}) \text{:>:} \text{C 'b'}) \end{aligned}$$

Semi-embedded DSL (with  $\text{compileReg} :: \text{Regex} \rightarrow \text{Q Exp}$ ):

$$\text{isAsB} = \$ (\text{compileReg} (\text{Rep} (\text{C 'a'}) \text{:>:} \text{C 'b'}))$$



# Building block 1: Template Haskell

Template Haskell [Sheard & Peyton-Jones Haskell'02] blurs the edges between embedded and semi-embedded

Embedded DSL (deep embedding):

$$\begin{aligned} \text{runReg} &:: \text{Regex} \rightarrow \text{String} \rightarrow \text{Bool} \\ \text{isAsB} &= \text{runReg} (\text{Rep} (\text{C 'a'}) :>: \text{C 'b'}) \end{aligned}$$

Semi-embedded DSL (with  $\text{compileReg} :: \text{Regex} \rightarrow Q \text{Exp}$ ):

$$\text{isAsB} = \$ (\text{compileReg} (\text{Rep} (\text{C 'a'}) :>: \text{C 'b'}))$$

Unlike *run*, *compile* provides static error handling and (programmer defined) optimisation.

## Building block 2: Quasi-quoters

Quasi-quoters [Mainland, Haskell '07] blur the edges between embedded and standalone

Embedded Language:

$$isAsB = runReg (Rep (C 'a') :>: C 'b')$$

Standalone language (imaginary):

$$isAsB = a * b$$

## Building block 2: Quasi-quoters

Quasi-quoters [Mainland, Haskell '07] blur the edges between embedded and standalone

Embedded Language:

$$isAsB = runReg (Rep (C 'a') :>: C 'b')$$

Semi-embedded language (with `regex :: QuasiQuoter`):

$$isAsB = $(compileReg [regex | a* b |])$$

Standalone language (imaginary):

$$isAsB = a* b$$

## Building block 3: BNFC

BNFC [Forsberg&Rante, Haskell '04] is a compiler front end generator with a standalone grammar DSL, Labelled BNF.

```
Stm ::= Ident "=" Expr ";" ;  
Stm ::= "while" "(" Expr ")" "{" [Stm] "}";  
Expr0 ::= Expr1 "<" Expr1 ;  
Expr1 ::= Expr1 "+" Expr2 ;  
Expr2 ::= Ident ;  
Expr2 ::= Integer ;  
  
Expr ::= Expr0 ;  
Expr0 ::= Expr1 ;  
Expr1 ::= Expr2 ;  
Expr2 ::= "(" Expr ")";
```

## Building block 3: BNFC

BNFC [Forsberg&Rante, Haskell '04] is a compiler front end generator with a standalone grammar DSL, **Labelled** BNF.

```
SAss.   Stm ::= Ident "=" Expr ";" ;
SWhile. Stm ::= "while" "(" Expr ")" "{" [Stm] "}";
ELt.    Expr0 ::= Expr1 "<" Expr1;
EPlus.  Expr1 ::= Expr1 "+" Expr2;
EVar.   Expr2 ::= Ident;
EInt.   Expr2 ::= Integer;
..       Expr ::= Expr0;
..       Expr0 ::= Expr1;
..       Expr1 ::= Expr2;
..       Expr2 ::= "(" Expr " )";
```

# Our contribution: BNFC-meta

BNFC-meta is the semi-embedded counterpart of BNFC:

*bnfc* :: *Grammar* → *Q* [*Dec*]

```
import Language.LBNF
```

```
bnfc [lbnf]
```

```
SAss. Stm ::= Ident "=" Expr ";"
```

```
SWhile. Stm ::= "while" "(" Expr ")" "{" [Stm] "}"
```

```
ELt. Expr0 ::= Expr1 "<" Expr1
```

```
EPlus. Expr1 ::= Expr1 "+" Expr2
```

```
EVar. Expr2 ::= Ident
```

```
EInt. Expr2 ::= Integer
```

```
.. Expr ::= Expr0
```

```
.. Expr0 ::= Expr1
```

```
.. Expr1 ::= Expr2
```

```
.. Expr2 ::= "(" Expr ")"
```

```
]
```

## While-language example continued

```
*Main> :i Expr
data Expr = ELt Expr Expr
          | EPlus Expr Expr
          | EVar Ident
          | EInt Integer
```

## While-language example continued

```
*Main> :i Expr
data Expr = ELt Expr Expr
          | EPlus Expr Expr
          | EVar Ident
          | EInt Integer
```

```
*Main> pStm $ tokens "while (a < 10) {a=a+1;}"
Ok (SWhile (ELt (EVar (Ident "a")) (EInt 10))
 [SAss (Ident "a") (EPlus (EVar (Ident "a")) (EInt 1))])
```



## While-language example continued

```
*Main> :i Expr
data Expr = ELt Expr Expr
          | EPlus Expr Expr
          | EVar Ident
          | EInt Integer
```

```
*Main> pStm $ tokens "while (a < 10) {a=a+1;}"
Ok (SWhile (ELt (EVar (Ident "a")) (EInt 10))
 [SAss (Ident "a") (EPlus (EVar (Ident "a")) (EInt 1))])
```

```
*Main> putStrLn $ printTree (SWhile (ELt (EVar ...
while (a < 10){
  a = a + 1 ;
}
```

## Advantages of BNFC-meta over BNFC

- No need to depend on command line tools!

## Advantages of BNFC-meta over BNFC

- No need to depend on command line tools!
- The abstract *Grammar* type is now exposed to the user
  - We can write several useful combinators for grammars
  - LBNF can use the Haskell module system

BNFC-meta is ...

- ... an Embedded (Parser Generator)  
an example of a semi-embedded language.

BNFC-meta is ...

- ... an Embedded (Parser Generator)  
an example of a semi-embedded language.
- ... an (Embedded Parser) Generator  
a tool for developing other semi-embedded languages.

# BNFC-meta generates quasi-quoters

In the while-language example:

```
*Main> :i stm  
stm :: QuasiQuoter
```

# BNFC-meta generates quasi-quoters

In the while-language example:

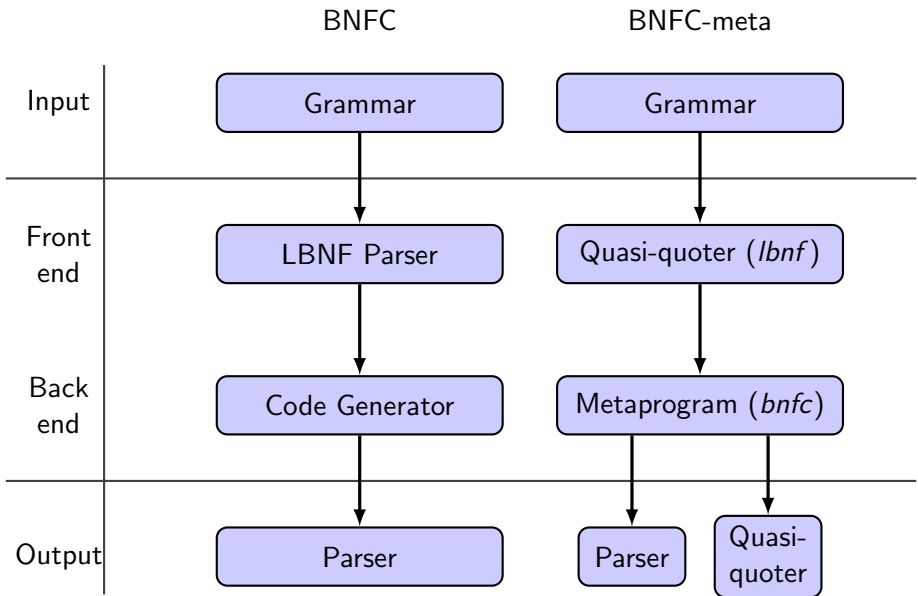
```
*Main> :i stm
stm :: QuasiQuoter
```

In another module we can import *stm*.

```
wait10 = [stm |
           while (a < 10) {
             a = a + 1;
           }
         |]
```

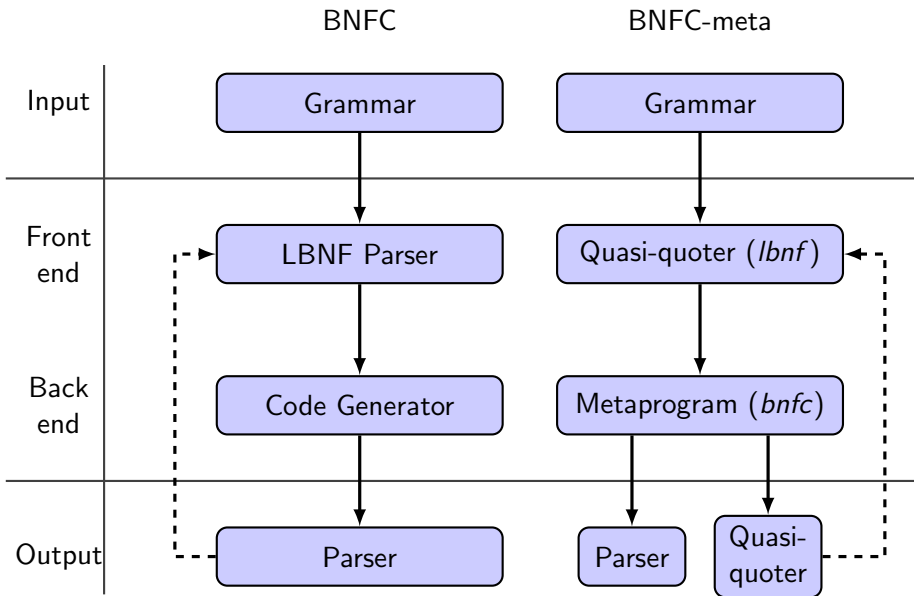
```
*Main> wait10
SWhile (ELt (EVar (Ident "a"))) (EInt 10))
[SAss (Ident "a") (EPlus (EVar (Ident "a"))) (EInt 1))]
```

# Bootstrapping





# Bootstrapping



# BNFC-meta adds anti-quoting to LBNF

```
import Language.LBNF
bnfc [lbnf |
  SAss.  Stm  ::= Ident "=" Expr ";"
  SWhile. Stm ::= "while" "(" Expr ")" "{" [Stm] "}";
  ELt.   Expr0 ::= Expr1 "<" Expr1;
  EPlus. Expr1 ::= Expr1 "+" Expr2;
  EVar.  Expr2 ::= Ident;
  EInt.  Expr2 ::= Integer;
  ..     Expr  ::= Expr0;
  ..     Expr0 ::= Expr1;
  ..     Expr1 ::= Expr2;
  ..     Expr2 ::= "(" Expr ")";
  $.     Expr2 ::= "%" Ident;  -- Anti-quoting spec.
  |]
```

# BNFC-meta Anti-quoting

We can define an anti quoting operator with a special '\$' label

In the while-language grammar:

```
$ . Expr2 ::= "%" Ident;
```

# BNFC-meta Anti-quoting

We can define an anti quoting operator with a special '\$' label  
In the while-language grammar:

```
$. Expr2 ::= "%" Ident;
```

In the client code:

```
wait steps = [stm |  
  while (a < %steps) {  
    a = a + 1;  
  }  
  ]
```

```
*Main> :i wait  
wait :: Expr -> Stm
```

# BNFC-meta Anti-quoting

We can define an anti quoting operator with a special '\$' label  
In the while-language grammar:

```
$. Expr2 ::= "%" Ident;
```

In the client code:

```
wait steps = [stm |  
while (a < %steps) {  
  a = a + 1;  
}  
|]
```

```
*Main> :i wait
```

```
wait :: Expr -> Stm
```

```
*Main> putStrLn $ printTree $ wait [expr| b+1 |]
```

```
while (a < b + 1){
```

```
  a = a + 1 ;
```

```
}
```

Other noteworthy features:

- Generated quoters can define patterns:  
 $eval :: Exp \rightarrow Int$   
 $eval [exp \mid \%a + \%b \mid] = eval\ a + eval\ b$
- Works well for large grammars, tested on ANSI C
- Useful by-products: alex-meta, happy-meta
- It is available on Hackage  
`cabal install BNFC-meta`

We use metaprogramming both to embed parser generators and to generate embedded parsers.

- Embedded (Code Generator)
  - Haskell code generators (happy, alex, BNFC) can be made into libraries (\*-meta)
  - Easier to use and less error prone
  - Users can access the abstract syntax of the input DSL
  - Quasiquoters are used to support existing DSL syntax
- (Embedded DSL) Generator
  - BNFC-meta supports development of embedded languages
  - Generates regular parsers and quasi-quoters
  - Anti-quoting can be specified directly in the grammar: abstract and concrete syntax can be mixed seamlessly

## Extra example

```
{-# LANGUAGE QuasiQuotes, TemplateHaskell #-}  
module RegexGrammars where  
import Language.LBNF  
import Language.LBNF.Grammar  
minimal :: Grammar  
minimal = [lbnf |  
  RAlt. Reg1 ::= Reg1 "|" Reg2;  
  RSeq. Reg2 ::= Reg2 Reg3;  
  RStar. Reg3 ::= Reg3 "*";  
  REps. Reg3 ::= "eps";  
  RChar. Reg3 ::= Char;  
  
  ..Reg ::= Reg1;  
  ..Reg1 ::= Reg2;  
  ..Reg2 ::= Reg3;  
  ..Reg3 ::= "(" Reg ")";  
|]
```



# Simple grammar combination

```
-- example continued ...  
combine :: Grammar → Grammar → Grammar  
combine (Grammar i) (Grammar j) = Grammar (i ++ j)  
extended :: Grammar  
extended = combine minimal [lbnf |  
  RPlus.   Reg3 ::= Reg3 "+";  
  ROpt.   Reg3 ::= Reg3 "?";  
  ]
```

# Program transformation with surface syntax

$topdown :: (Reg \rightarrow Reg) \rightarrow Reg \rightarrow Reg$

$topdown f rx = \mathbf{case} f rx \mathbf{of}$

$[reg \mid \%e1 \quad \% e2 \mid ] \rightarrow [reg \mid \{r e1\} \quad \{r e2\} \mid ]$

$[reg \mid \%e1 \mid \% e2 \mid ] \rightarrow [reg \mid \{r e1\} \mid \{r e2\} \mid ]$

$[reg \mid \%e1 * \quad \mid ] \rightarrow [reg \mid \{r e1\} * \quad \mid ]$

$[reg \mid \%e1 + \quad \mid ] \rightarrow [reg \mid \{r e1\} + \quad \mid ]$

$[reg \mid \%e1 ? \quad \mid ] \rightarrow [reg \mid \{r e1\} ? \quad \mid ]$

$e \quad \rightarrow e$

**where**  $r = topdown f$

$transform :: Reg \rightarrow Reg$

$transform = topdown step \mathbf{where}$

$step rx = \mathbf{case} rx \mathbf{of}$

$[reg \mid \%e \mid eps \quad \mid ] \rightarrow [reg \mid \%e ? \quad \mid ]$

$[reg \mid \%e1 \quad \% e2 * \mid ]$

$\mid e1 \equiv e2 \quad \rightarrow [reg \mid \%e1 + \mid ]$

$\mid otherwise \quad \rightarrow rx$

$e \quad \rightarrow e$