

PFront: a language with extensible syntax

Meta Alternative Ltd.

Apr 2009

1 Introduction

Programming languages derived from Lisp are widely used for metaprogramming. Having a single fixed syntax which maps directly to the language's abstract syntax tree allows to change language semantics dynamically. But this approach has several drawbacks which prevent it from dominating the software development industry. The lack of syntax variations is the most problematic of them.

With the recent resurrection of Parsing Expression Grammars (PEGs), the dream of a truly extensible programming language found its way to practical implementations. One of the inspiring works in this area is an experimental language Katahdin. It is based on PEGs and provides a basic form of metaprogramming. But to make this approach really practical, a much more flexible metaprogramming support is required, and this is precisely what MBase framework is designed for.

Here we will introduce a PFront extension language on top of MBase framework. It is based on a flexible PEG grammar and contains all the features needed to extend its own syntax and semantics, while retaining all of the underlying MBase abilities for implementing programming languages. Technically, PFront is just a syntax front-end for MBase.

PFront is part of forthcoming 1.0 release of MBase framework.

2 Setting the scene

PFront syntax is similar to all other curly-brackets languages (like C, Java, ECMAScript, etc.). The most significant difference is that it is expression-oriented, and there is no distinction between statements and expressions. Here we will not describe PFront (and the underlying MBase) syntax and semantics in detail, but instead will immediately start with some practical examples, expecting some level of familiarity with any of the curly-brackets languages from the reader.

Our first example will demonstrate how to define a simple iterative construction on top of a functional language which originally does not contain anything relevant. PFront allows to define loops in a form of a local recursive function, like in the following piece of code:

```
do myloop ( i = 0 )
{
  if(i<10) {
    print(i); myloop(i + 1);
  }
  else { println("Done."); }
}
```

A local recursive function is a very powerful language feature, but in this trivial case it is an obvious overkill. So it may be beneficial to amend the PFront

syntax to allow some simpler iterative constructions¹.

The following code will introduce a new syntax into our language:

```
syntax in expr, start:
    'for "(" [qident]:i "=" [expr]:init ";" [expr]:cl ";" [expr]:step "'
      [expr]:body'
{
    nm = gensym();
    nxt = '\nm\ ( \step\ )';
    'do \nm\ ( \i\ = \init\ ) { if \cl\ { \body\; \nxt\ } }'
}
```

And what is particularly exciting — we can use this new syntax immediately, in the same source file:

```
{ for(i=0;i<10;i+1) { print(i); println("Done."); }
```

Now, what exactly does the definition above mean? A special form “**syntax in** *where*” declares a location in the PFront grammar where a new syntax should be added. It is possible to insert bits of grammar into various parts of the expression syntax, including different priority levels of infix operations, top level expressions syntax, etc. A complete initial PFront grammar and AST are available for reference, so the user will always know the consequences of amending the grammar.

In the next line a syntax itself is defined, using the simplified PEG. Simple words like “for” are treated as keywords. Strings (like “(”, “)”, “;”, “=”) are lexical elements. Other PEG nodes are referenced as in “[*expr*]”, “[*qident*]”, etc. Node values may be bound to names, which are later referenced in the generator code.

Inside the AST generator code we are using a quasiquotation. Any valid PFront expression between back quotes is compiled into a code which generates an AST of this expression. Inside certain parts of expression (where identifiers and expressions can be referenced), an unquoting can be used. Unquoting is a name of a variable between back slashes. A value of a named variable is substituted into an AST.

PFront compiler will expand the example above into something like this:

```
{ do Z123456 ( i = 0 ) {
    if ( i < 10 ) { print(i); Z123456(i+1); }
}; println("Done.");
}
```

The same technique may be used for defining **unless** syntax:

¹Do not be afraid of the recursion costs, the underlying MBase compiler will unroll it into a local iteration any way.

```

syntax in expr, start:
  'unless "(" [expr]:cnd ")" do [expr]:body'
{
  nm = gensym();
  'do \nm\ () { if \cnd\ then nil else {\body\;\nm\ () } }'
}

```

It is also possible to amend infix grammar entries. For example, the following code will add the string concatenation operator:

```

syntax in expr, bin2:
  '[basicexpr]:a "@" [eterm]:b '
{
  '%L[;S<<:;]( \a\,\b\ )'
}

```

A special syntax for identifiers with forbidden characters is used here in order to generate the MBase expression “(S<< a b)”.

3 Embedding a complex parser

Of course, one line grammars are very limited. A “real” parser must be defined in case there is a need to embed a significantly different language into PFront syntax.

In the following example we will define a simple grammar for S-expressions, going back to Lisp roots.

```

parser sexpr ( pfront ) {
  // All the tokens will ignore leading Spaces:
  !!Spaces;
  sexpr := sxnde;
  sxnde := { "(" " " => $nil() }
    / { "(" [sxndx]:x " " => x }
    / { ",", "@" [sxnde]:a => "unquote-splicing"(a) }
    / { ",", " " [sxnde]:a => "unquote"(a) }
    / { "\" " [sxnde]:a => "quasiquote"(a) }
    / { [tQUOTE] [sxnde]:a => "quote"(a) }
    / { [ident]:s => s }
    / { [string]:s => s }
    / { [number]:n => n }
  ;
  sxndx := { [sxnde]:a "." [sxnde]:b => $cons(a,b) }
    / { [sxnde]:a [sxndx]:b => $cons(a,b) }
    / { [sxnde]:x => $wrap(x) }
}

```

```
};
```

Here we can see another useful feature of PEGs — grammars may be inherited. Our “sexpr” grammar inherits “pfront”, which is the core PFront grammar. In fact, we only use definitions for a couple of tokens from it: “ident”, “string”, “number”, “tQUOTE” and “Spaces”, but terminals and macros could be referenced as well. Rules are not inherited.

Now this syntax can be plugged into PFront:

```
syntax in expr, start ( sexpr ): ' "#" [sxnde] :e '  
{  
  ['lisp';e]  
}
```

We can now embed a Lisp code anywhere in PFront:

```
{  
  a = 2;  
  #(writeline (%L[;+;] a a));  
  writeline(#(fromto 1 10));  
}
```

Note that “#” is highlighted now as a lexical element. A literate programming tool, which was used to generate this article, is using the same parsing engine, and it is sensitive to all users’ changes to the grammar.

4 AST support and pattern matching

The true power of MBase is not just in metaprogramming support. It is rooted in the practical selection of language functionality designed for typical tasks of programming languages implementation. Most important are pattern matching and abstract syntax trees support. Core PFront defines special syntax for these features.

4.1 Simple eDSL

In the following example we will define a simple language, a compiler and an interpreter for it.

An AST declaration is quite similar to algebraic data types in languages like ML or Haskell:

```
ast calc {  
  expr =
```

```

    plus(expr:a,expr:b) | minus(expr:a,expr:b)
    | mul(expr:a,expr:b) | div(expr:a,expr:b)
    | apply(var:a, *expr:b) | const(number:v);
var is string:s;
}

```

The language above is made of a single recursive expression type. In order to compile it to PFront AST, we can use the quasiquotation syntax:

```

function mycompile(c)
{
  visit:calc(expr: c) {
    deep expr {
      plus → ‘\a\+\b\’;
      minus → ‘\a\-\b\’;
      div → ‘\a\ / \b\’;
      mul → ‘\a\*\b\’;
      apply → (‘call’ : ([‘var’;a] : b))
      const → [‘number’;v]
      else { ‘0’ }
    }
  }
}

```

Here we’ve met the limits of the quasiquotation for the first time, in constructing a numeric constant and a function application of an arbitrary number of arguments. When this rare thing happens, we can always construct AST directly (and this is why an AST definition is available as part of PFront users manual).

The “visit” syntax is an easy wrapper over the underlying MBase AST visitors macro. With the “deep” rule it generates the code which traverses the AST (depth first), replacing each node with the value produced by the right side of “→”. Effectively, this visitor replaces all of the “calc” AST with a corresponding PFront AST.

```

writeline(
  inplace(
    mycompile(
      [‘plus’;[‘mul’;[‘const’;2];[‘const’;2]];
      [‘mul’;[‘const’;3];[‘const’;3]])))

```

Here, “inplace” is a macro which expands into its argument value. It is quite useful for constructing code in compilation time without defining a dedicated macro.

The interpreter for the same language is surprisingly similar to the compiler:

```

function myinter(c)
{
  visit:calc(expr: c) {
    deep expr {
      plus → a+b
      | minus → a-b
      | div → a/b
      | mul → a*b
      | apply → applyfunction(a, b)
      | const → v
      | else { 0 }
    }
  }
}

```

Now, to illustrate pattern matching, we will map PFront AST to this toy language AST:

```

syntax in expr, start: 'mylang "<" [expr]:code ">"
{
  e1 = do loop (e = code)
  {
    match e with
      ('call':(var('apply')):(var(fn):args))
        → 'apply':(fn:(map a in args do loop(a)))
      | ('call':(var(v):args)) → v:(map a in args do loop(a))
      | number(n) → ['const';n]
      | other → cerror(["Wrong!";e])
    };
  return ['number';myinter(e1)];
}

```

And the usage is as follows: `writeline(mylang<plus(mul(2,2),mul(3,3))>)`

It is also worth mentioning that a user-defined syntax may return a different AST, not necessarily compatible with the PFront core.

4.2 Factorial expression

This article would not be complete without a classic example — a factorial function. The definition is trivial:

```

function fact(n)
{
  do ifact(n = n, c = 1)
  {
    if (n>0) ifact(n-1,c*n) else c
  }
}

```

```
}  
}
```

And, of course, a special syntax may be defined:

```
syntax in expr, bin2:  
    ' [basicexpr]:e "!" '  
{  
    match e with  
        number(n) → ['number'; fact(n)]  
    | other → 'fact( \e\ )'  
}
```

Now for a constant expression like “*print(10!)*”, a constant factorial value will be substituted. For any other expression types, like in “*print({a=5; (a*2)!})*”, a function call is substituted.

This article has covered only a small fraction of PFront features. There is a special syntax for more complex “one line” grammars, syntax highlighting annotations, text editor and pretty-printer interaction control, some predefined embedded languages (including a low-level Not.Net language with a syntax similar to C#). And a user can add any syntactic and semantic properties to the language, utilising this combination of extendible grammar and metaprogramming.