# 1 Introduction

PFront is one of the MBase programming languages. A typical MBase language adds new functionality to MBase framework, new semantic features which can be re–used within any other MBase language. PFront is different, it does not add any new semantic blocks, but instead it is an extensible syntax front-end to all of the existing and possible MBase languages. It is intended to be a common base for designing user–friendly DSLs with an arbitrary syntax.

As any other MBase component, PFront is highly extensible. Here we will describe its core and some of the default extensions.

# 2 Expressions

PFront functional core is mostly expression–based. It means that constructions like "`if` something `then` something", which are dedicated "statements" in a majority of modern programming languages, are of the same sort as constructions like "$a + b$". Expressions always yields a value, while statement performs an action.

There are several groups of expressions in PFront:

- *Constants*: symbols, numbers, strings, characters, etc.
  - Symbols: `'a'`, `'a-b-c'`, `'+'`, `'a123-456'`
  - Numbers: `123`, `-600`
  - Strings: `"Hello, World"`
  - Characters: `'A'c`, `7c`, `' 'c`
  - Nil: $\emptyset$
  - Logical: `true`, `false`
- *Variables*: *abc*, *ACamelCaseVariable*, *%L[;-mbase-*-variable-;]*
- *Constructors*:
  - list constructor: [1;2;3]
  - single cons: `1` : $\emptyset$
  - lists concatenation: [1;2;3] $\oplus$ [4;5;6]
  - integer range: [`1` .. *n*]
- *Binary operators* with precedence: `1` + `2`, *a***b*+*c***d*, `1` < `2`
- *Explicit precedence*: (`2`+`2`)*`2`
- *Function application*: *print*(`"Hello, World!"`), *nth*(2,[1;2;3;4]), *quit*()
- *Return expression*: `return` *x*

  *Please note that it is for a better source code formatting only. Return expression yields a value of its argument, and it does not break the execution of the current code block — it is different from the behaviour of* `return` *in languages like C#*
- *Binding* (makes sense within a sequence only): *str* = `"Hello, World!"`
- *Sequence*: {*a* = `2`; *b* = `2`; `return` *a***b*}
- *Functional abstraction*: $\lambda$(*a*,*b*) {*a***a*+*b***b*}
- *Selection*: `if`(*a*>*b*) *print*(`"A!"`) `else` *print*(`"B!"`)
- *Local recursive function*[1]: `do` *fact*(*n* = `5`) {`if` (*n*>`0`) *n***fact*(*n*-`1`) `else` `1`}

A normal evaluation order for expressions is left to right.

There is also a number of compound expressions pre–defined in the PFront core. Some of them will be discussed later.

**N.B.:** Please see the MBase documentation for a reference on all the standard functions and macros.

[1] This is the same concept as the Scheme "named let" construction

# 3 Top–level statements

PFront is *mostly* expression–based, but there is a number of constructions that are only valid in a top–level context. These constructions includes function and macro definitions, syntax declarations, parsers, AST specifications.

A function definition is similar to the functional abstraction, with the only difference that it binds a function to a given name:

```
function add(a,b) { return a + b; }
```

Function defined this way is globally visible. It is, however, possible to define a locally–scoped function. One way is to use local recursive function syntax, another is to bind a value of a functional abstraction to a variable.

For example:

```
function test(n) {
  f1 = λ(a,b) { return a*a+b*b;};
  f2 = λ(x) { return f1(x,x); };
  return f2(n);
}
```

Top–level expressions can be grouped, which is useful for code generation:

```
{
 function a() { print("A"); }
 {
   function b() { print("B"); }
   function c() { print("C"); }
 }
}
```

# 4 Lists

PFront, as well as the underlying MBase core, is a dynamically typed language. In practice it means that variables, function arguments and list elements could be of any object type. Value types are wrapped into objects. Several types are specifically supported by the language: strings, symbols (`Meta.Scripting.Symbol` instances), integer numbers (`System.Int32`), cons cells (`Meta.Scripting.Pair` instances). The latter forms so called "lists", data structures common to all the languages descending from the Lisp origin. Syntax for representing lists in PFront is different from the standard S–expressions[2].

"Lists" in PFront are trees made of binary cons cells, or "pairs". The most basic operation is constructing a pair: *head* : *tail*. Head is the left element of a pair, and tail is the right element of a pair. By convention, a "flat" list is made of a sequence of tail–attached pairs with list elements attached to pairs heads. For example, a list [1;2;3] can be generated as 1 : ( 2 : ( 3 : ∅)) . Here, ∅ denotes a null pointer, or "nil" in a Lisp terminology.

Special case list constructor for lists starting with a symbol has the following syntax: `'tag'(1,2,@rest)`. Values of special case constructor arguments are substituted as flat list elements. If a value is prefixed with `@`, the value is appended. For example, {*l* = [1;2;3]; `'abc'(0,@`*l*`,4)`} will give the list `['abc';0;1;2;3;4]`.

Lists are very flexible, so most data structures used by PFront are actually represented as lists. Most important use of lists in PFront is representing Abstract Syntax Trees and similar intermediate structures.

Lists are constructed (e.g. by parsers, or during AST transforms) and deconstructed (when an access to specific list element is needed). In following sections we will introduce PFront special features for dealing with lists in an easy and efficient way.

# 5 Pattern matching

One form of list deconstruction is pattern matching. For example, for a list *l* = `['add';2;2]` we can get second and third elements using the following construction:

[2]But of course it is possible to use the S–expressions syntax as well

```
match l with ['add';a;b] ↦ a + b
```

Here values of second and third elements of a list was bound to variables $a$ and $b$ and used within expression $a + b$.

For this special kind of lists, where the first element is a symbol, there is a somewhat shorter form of patterns available:

```
match l with add(a,b) ↦ a + b
```

It is possible to match against a number of patterns. For example, the following code iterates over a flat list elements and performs some action with each of them, and stops once it meets a symbol `'stop'`:

```
do my_iter ( l = lst )
  match l with
    ∅ ↦ ∅
  | 'stop' : tail ↦ ∅
  | head : tail ↦ { dosomething(head); my_iter(tail) }
```

Valid patterns are: pairs and lists constructors, variables, symbols, strings and numbers.

There is a special syntax defined for matching against a single pattern ("format"):

```
format ( l : [a;[b;c]]) do_something(a,b,c)
```

Another syntax for the same construction is a format binding:

```
{
 <[a;[b;c]]> = l;
 do_something(a,b,c);
}
```

# 6 ASTs

The main purpose of MBase framework in general and PFront language in particular is in implementing programming languages. Almost all compilers and interpreters deals internally with abstract syntax trees, and most of the operations with trees are quite typical. There is a rich set of special constructions for defining abstract syntax trees structure and implementing transformations over these trees.

ASTs in MBase are somewhat similar to algebraic data types in languages like ML and Haskell.

```
ast calc {
  expr =
      plus(expr:a,expr:b) | minus(expr:a,expr:b) | mul(expr:a,expr:b)
    | div(expr:a,expr:b) | const(number:v);
}
```

Here, `expr` is a recursive data structure. AST nodes are represented as lists. For example, the following list conforms to the "calc" AST structure:

```
['plus';['const';2];['const';2]]
```

This sort of structure may originate from an expression parser, or be generated from some other structure via transformation.

One may wonder, why the precise structure definition for such a dynamically typed language? These definitions are used by *visitors* and *iterators* — constructions needed to implement AST transformations.

For example, a simple interpreter for the "calc" AST defined above will look like this:

```
visit:calc(expr: c) {
   deep expr {
     plus ↦ a + b
   | minus ↦ a - b
   | div ↦ a / b
   | mul ↦ a * b
   | const ↦ v
   | else 0
   }
 }
```

This solution is quite different from what would be typical for a language like ML. This evaluator is obviously recursive, but the recursion is implicit. PFront compiler knows the structure of "calc" AST nodes, and implicitly inserts depth–first recursion wherever "expr" node recursively refers to itself. The very same behaviour is present for a multitude of node types (e.g., if there are separate expression and statement nodes), which allows to write much more dense and readable code than the explicit recursive pattern matching against algebraic data types.

Of course, sometimes an explicit recursion is necessary. For example, if we modify the AST above as follows:

```
ast calc {
  expr =
      plus(expr:a,expr:b) | minus(expr:a,expr:b) | mul(expr:a,expr:b)
    | div(expr:a,expr:b) | const(number:v)
    | var(ident:nm)
    | let(ident:varnm,expr:v, expr:body);
}
```

Here, "let" nodes introduces a context — a variable is bound to a value, so, this context is propagated upside down, whereas the default "deep" behaviour of PFront AST visitors is to traverse the tree downside up.

We have to rewrite our interpreter as follows:

```
do interpreter ( c = expr, env = ∅ )
  visit:calc (expr : c) {
    once expr {
     | let ↦ { newenv = [varnm;interpreter(v, env)]:env;
               interpreter(body, newenv);
             }
     | deep ↦
       {
         var ↦ %L[;lookup-env-car;](env,nm)
       | plus ↦ a + b
       | minus ↦ a - b
       | div ↦ a / b
       | mul ↦ a * b
       | const ↦ v
       | else 0
       }
    }
  }
```

Here, we added an explicit recursion only where it is needed, where an additional argument must be passed to the current visiting environment.

"once" visiting strategy does not go into any sub–nodes of the current node, which is different from the "deep" strategy. But it is possible to have "once" strategy for only a selection of node variants, and proceed with the "deep" way with all the rest — this is precisely what the construction `| deep -> { ... }` does.

Please note — due to this syntax, variant tags can't be neither `else` nor `deep`. All other symbols are valid.

There are several standard patterns of tree transformations, and majority of interpretation and compilation tasks falls into a combination of them. Most common patterns are explained in the appendix.

### 6.1 AST definitions

AST definition is a top–level statement. The syntax is following:

`ast name [(inheritance-map, ...)] { node; ... }`

Node definition is either a structure or a variant. Structures are defined as follows:

`nodename is (format...)`

Variant definition syntax is following:

`nodename = var1(format...) | var2(format...) | ...`

Format is:

`element, format` or `element , . format`

Element is:

`[*] nodetype:name`

Here, $*$ stands for 0-or-more nodes of a given type. Nodetype may be a reference to a node defined in this AST, or any other symbol.

## 7 Quasiquotation

In PFront most ASTs ends up in a generated PFront or lower level MBase code. To simplify the generation of PFront (and all the derived languages) ASTs, there is a special syntax for quasiquotation.

For example, the expression ‘*a + b*‘ has a value `['binop';'+';['var';'a'];['var';'b']]`. This is a PFront core AST, which is consequently translated into an underlying MBase representation.

It is possible to substitute a value of a variable at certain placeholders within a quasiquotation expression. The interpreter above can be turned into a compiler easily:

```
visit:calc(expr: c) {
   deep expr {
     plus ↦ ‘\a\ + \b\‘
   | minus ↦ ‘\a\ - \b\‘
   | div ↦ ‘\a\ / \b\‘
   | mul ↦ ‘\a\ * \b\‘
   | const ↦ 'number'(v)
   | var ↦ 'var'(nm)
   | let ↦ ‘{ \varnm\ = \v\; return \body\ }‘
   | else { ‘0‘ }
   }
 }
```

It is even easier than the interpreter — since we can use “deep”–only strategy again.

This compiler generates a PFront core AST. Next sections will give a hint on what to do with this ASTs further.

## 8 Parsers

PFront mainly focus on syntax, whereas the rest of MBase is responsible for semantics. Parsing in PFront is implemented on top of Parsing Expression Grammars. The most appealing property of PEGs is the absence of a separate lexing stage, which means that it is possible to mix languages with incompatible sets of lexemes.

### 8.1 One–line syntax extensions

PFront syntax is extensible. There is a number of entries in the core syntax where additional parsers may be plugged in.

For example, we want to define a string concatenation binary operator. It should be plugged into the “expr”, section “bin2”, which is the same precedence as arithmetic operators `*` and `/`. The definition is as follows:

```
syntax in expr, bin2: ' [basicexpr]:a "++" [eterm]:b '
{
 return ‘%L[;string-append;]( \a\ , \b\ )‘
}
```

Here we can see another benefit of not having a separate lexing pass: a new operator “++” can co–exist with the previously defined operator “+”. Now we can use expressions like `"Hello, world" ++ "!"`.

An obligatory factorial example:

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

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

Now we can use expressions like x! or 10!. The latter would be evaluated in compilation time.

## 8.2 Mini–PEG language

Expressions between single quotes in those "syntax in" examples above belong to the miniPEG language, which is a detached part of PFront.

All the standard PEG expressions are supported:

- `e+` — one ore more
- `e*` — some
- `e?` — none or one
- `!e` — not
- `&e` — and the rest is
- `e1 e2` — sequence
- `e1 / e2` — ordered alternative
- `( e )` — grouping

Specific syntax elements are:

- `[node]` — a reference to a named node (token, terminal or a regular expression)
- `e : name` — bind the value of `e` to a variable `name`. An expression here can only be a node reference or a macro application.
- `macro<e1, e2, ...>` — substitute a macro. Argument expressions should not contain any bindings and anonymous nodes
- `{ e => constr }` — anonymous node with constructor. Anonymous nodes are always of a "terminal" type, with all the rules applied.

Atomic (or "trivial") expressions are:

- `.` — any character
- *number* — an ASCII code for a character
- `[A-Z]` — a character range (from character A to character Z inclusive)
- `"string"` — a sequence of characters, if inside a terminal node — a "lexical" rule is applied
- `ident` — a sequence of characters, with a "keyword" rule applied if in a terminal node
- `[rule : "string"]` — a sequence of characters with a given rule applied

And finally, top level definitions — nodes, rules and macros:

- `[`*rulename*`:]` ⇐ `expr` ⇒ *constr*;

  Defines a rule. Expression template must contain at least one reference to a node, which name is `rulename`. Every new string which falls under a given rule will generate a new node, formed of a given template.

  It is important to define at least two built–in rules — `lexical`, which applies to all the simple strings found inside terminal nodes, and `keyword`, which applies to all the identifier–like entries found inside terminal nodes. "One–line" syntax extending expressions are always treated as terminals.

  A typical example of a keyword rule definition is following:

  ```
  [keyword:] ⇐ [keyword] ![IdentRest] ⇒ {ctoken = keyword};
  ```

  Here, `IdentRest` should recognise any character which may be found within an identifier lexeme. This rule means that, for example, if we generate a keyword `int` as a part of some syntax, a string `"integer"` won't be recognised as keyword `int` followed by identifier `eger`. A constructor part in this example hints the syntax highlighting of all the keywords for text editor and literate programming tools.

  Nodes raised by rules are always of a "token" type, which means they're skipping the currently defined ignorance node instances in the beginning.

- `!!Nodename;`

  Defines a node to be ignored at a beginning of all the *tokens*. Typically, it would be something like `!!WhiteSpacesOrCommen`

- `&Nodename;`

  Defines an empty placeholder node. This sort of nodes is necessary for dynamic extensions.

- `Nodename<arg1,arg2,...> := expr => constr;`

  Defines a macro with a given name and a list of arguments. Template expression may contain references to either `Nodename`, which will be a recursive self–reference for any macro instantiation, or to its arguments, which will be substituted on instantiation.

  A typical usage example:

  ```
  cslist<e,s> ⇐ { [e]:head [b] [cslist]:tail ⇒ $cons(head,tail) }
              / { [e]:head ⇒ $wrap(head) };
  ```

  This macro can be used to define a something–separated–list of elements, for example, in a following syntax:

  ```
  function [ident]:name "(" cslist<[ident],","">:args ")"
  ```

  Please note that only the value of the whole macro is bound to the `args` variable, not the individual arguments.

- `@@Nodename := expr;`

- `@@Nodename := expr => annotation;`

  Defines a "regular expression" node. It does not construct any value, so only annotation part of a constructor is allowed. No rules are applied, strings and identifiers are treated as is. Nodes of this type are not memoised by the Packrat parser backend.

- `@Nodename := expr;`

- `@Nodename := expr => annotation;`

  Defines a "token" node. Token nodes always construct tagged string values, so no specific constructors are allowed, annotations only. No rules are applied inside token nodes. Ignoration rule is applied.

- `Nodename := expr;`

- `Nodename := expr => constr;`

  Defines a "terminal" node, which may or may not contain constructors. Terminal nodes values are memoised by the Packrat backend by default.

- `binary Nodename := binexpr | binexpr ...;`

  Defines a Pratt binary expressions parser node. Please note that the delimiter is "`|`", instead of "`/`", to stress the difference between a Pratt node and a normal PEG sequence.

  Here, `binexpr` is one of the following:

  - `( Precedence Associativity ) BinNode OpExpr BinNode => constr;`

    Adds a binary expression with a given numeric precedence and associativity (`left`, `right` or nothing, defaults to left). `BinNode` must always refer to the `Nodename` of the same binary parser node. `OpExpr` can be any PEG expression.
  - `expr`

    A single expression must be given at the end of the list of binary variants.

Annotations are simply pairs of names and values (both conforming to identifier syntax), like in `{ctoken = keyword}`. This pairs guides syntax highlighting, identation, autocompletion, etc. Annotations does not affect parsing (i.e., values constructed by a parser).

Constructor syntax is following:

- `tag ( constr, constr, etc...)`

  Construct an AST node with a given tag and values.
- `$fun(constr, constr, etc...)`

  Apply a special two–way function to given values. See the predefined list of constructor functions below.
- `tag ...`
- `tag(...)`

  Create a constructor with a given tag, with all the named pattern arguments listed in an order of appearence. For example:

  `[expr]:a "+" [expr]:b => plus ...`

  is the same as

  `[expr]:a "+" [expr]:b => plus(a,b)`
- `...`

  Same as the one above, with a constructor tag taken from the name of a containing terminal node.
- `‘tag`

  Create a constant symbol
- `ident`

  Substitute a value of a variable

Tags here are either identifiers or `"strings"`, both are treated as symbols.

### 8.2.1 Extended one–line parsers

The `syntax in ...` construction with one–line parsers is powerful enough for most syntax extensions, but in some cases additional nodes should be defined. For example, if we need a small embedded regular expressions language, it can be defined as follows:

```
syntax in expr, start: ' "<" r ":" [regexp]:rgxp ">" '
+ {
    @reganychar ⇐ ![regspecchar] .;
    @@regspecchar ⇐ "("/")"/">"/"<"/"+"/"*"/"."/"["/"|";
    regexp ⇐
            { [regatom]:l [regexp]:r     ⇒ seq(l,r) }
          / { [regatom]:a "+"            ⇒ plus(a) }
          / { [regatom]:a "*"            ⇒ star(a) }
          / { [regatom]:l "|" [regexp]:r ⇒ or(l,r) }
          / { [regatom]:a                ⇒ a }
          ⇒ { qstate = pattern }
          ;
    regatom ⇐
            { "(" [regexp]:r ")"     ⇒ r }
          / { [charrange]:r          ⇒ r }
          / { "."                    ⇒ any() }
```

```
        / { [reganychar]:c          ⇒ char(c) }
        ;
  }
{
  return 'lisp'('compile_static_regexp'(rgxp));
}
```

Here, a one–line syntax is backed by a complex parser with two terminal, one token and one regular expression nodes.

### 8.2.2 Pre–defined functions

Constructor functions are special. They must be reversible, in order to generate pretty–printers automatically out of parsers. That's why not all the MBase and PFront functions are available.

- `$cons(a,b)` — same as $a : b$
- `$wrap(a)` — same as $[a]$
- `$nil()` — same as $\emptyset$
- `$list(a1, a2, ...)` — same as `[a1;a2;...]`
- `$symbol(str)` — convert a string to a symbol
- `$val(a)` — detach a value from a tagged token, convert it into a symbol. This one is specifically handled by the PEG compiler, in order to infere a possible tag
- `$sval(a)` — same as previous, but leaves a value as a string
- `$charcode(s)` — returns an integer ASCII code of a given character

## 8.3 Writing complex parsers

We've only discussed "syntax in" parsers before. But there is a generic PEG definition available as well, like in a following example:

```
parser helloworld ( ) {
  !!spaces;

  @@spaces ⇐ ([space] / [cr]) +;
  @@space ⇐ " "/[tab];
  @@cr ⇐ 13;
  @@tab ⇐ 9;

  name ⇐ ([A-Z]/[a-z]) ([a-z]+);

  @tkname ⇐ [name];

  helloworld ⇐ Hello "," [tkname]:nm "!" ⇒ nm;
}
```

And then, an expression `parse "Hello, world!" as` *helloworld* will return a value `(tkname . "world")`.

### 8.3.1 Left recursion

Left recursion is allowed in PEG, both direct and indirect. The obvious limitation is that only the node which is already left–recursive may be dynamically extended with a new left–recursive rule.

An example of a left recursive syntax:

```
parser calc (pfcommon) {
  !!Spaces;
  &expr_more; &term_more;
  expr ⇐ {[expr]:l "+" [expr]:r ⇒ plus(...)}
       / {[expr]:l "-" [expr]:r ⇒ minus(...)}
       / {[expr]:l "*" [expr]:r ⇒ mul(...)}
```

```
    / {[expr]:l "/" [expr]:r ⇒ div(...)}
    / [expr_more]
    / { "(" [expr]:e ")" ⇒ e }
    / [term]
    ;
  term ⇐ {[ident]:v ⇒ var(v)} / {[number]:n ⇒ const(n)}
    / [term_more]
    ;
}
```

Here, `expr` is left–recursive already, and so we can extend this parser with additional left–recursive entries:

```
syntax of calc in expr, more: '[expr]:l "!"' { 'factorial'(l) }
```

# 9 Code generation

In order to utilise PFront to its maximum power, it is important to understand the way it is being translated into MBase, and how both languages interacts.

Essentialy, MBase is a dialect of Lisp programming language. It is more akin to the Scheme family, with several subtle differences: only integer arithmetics supported, no first class continuations, no enforced hygiene for macro expansion. A precise language definition is out of the scope of this document, so only the parts relevant to PFront interaction will be described here.

PFront language core is defined by three usual parts: an abstract syntax tree, a parser and a code generator, which is, basically, a transformation from the AST into MBase expressions. One have to understand all three parts in order to implement syntax and semantic extensions for PFront. The complete source code is provided for reference:

- `pcommon.peg`, `pfront.peg`, `minipeg.peg` — PFront parser components
- `ast.al` — an AST definition for PFront
- `codegen.al` — PFront → MBase code generator implementation
- `notnet.hl` — Not.NET language extension
- `extensions.hl` — a collection of useful PFront and Not.NET extensions

The code is mostly declarative and self–documented. The most important entry points are:

- `pfront_top` function: translates top level AST statement into MBase
- `pfront_expr` function: translates simple expression AST into MBase
- `hlevel` AST definition: the core PFront AST
- `topexpr` AST entry: top level PFront statements
- `expr` AST entry: normal PFront expressions
- `lisp` variant of `expr` AST entry: an MBase code to be passed verbatim through the code generation.
- `expr` term of `pfront` parser: a generic expression entry.
- `qident` term: an unquotable identifier.
- `bexpr` term: lowest priority binary operations, contains `expr_bin1` dynamic entry.
- `eterm` term: higher priority binary operations, contains `expr_bin2` dynamic entry.
- `basicexpr` term: all the normal expression entries. Contains `expr_start`, `expr_preatom` and `expr_end` dynamic entries.
- `topexpr` term: top level PFront statements (global definitions, etc.). Contains `top_start` and `top_middle` dynamic entries.

There are two main techniques which should help in understanding the AST structure and code generation. First is to use the quasiquotation to obtain a genuine AST for a given expression. For example, ‘map *a* in [1..10] do *a***a*‘ will give `(map a (range (number 1) (number 10)) (binop * (var a) (var a)))`. Second technique is to apply the appropriate code generation function to see the actual MBase code: *pfront_expr*(‘map *a* in [1..10] do *a***a*‘) will give #(foreach-map (*a* (fromto 1 10)) (* *a* *a*)).

Any combination of a PFront AST, a verbatim MBase code and an explicit compilation of an AST into MBase can be used within syntax extensions. For a better performance and cleaner code, it is recommended to separate any complicated processing from the parsing pass, which means construction of an MBase macro application code from the syntax extension, and then dealing with its contents from the macro definition. Here is an example:

```
syntax in top, start: ' regexp [qident]:nm "=" [miniexpr]:e ";"? '
{
  return 'expr'('lisp'('my_peg_regexp'(nm,e)))
}
```

Here, a verbatim MBase `my_peg_regexp` macro application was constructed, without doing anything to the contents of expression `e`. The actual expression is processed by the macro, still trivial in this case, but it can be of an arbitrary complexity, whereas a syntax extension should only construct simple lists out of its input:

```
macro my_peg_regexp(nm, e)
{
  symbols(nn,nn1,nn2) {
   #‘(top-begin
       (packrat-ast ,nm ( )
          (terminal normal ,nn ,e (() ()))
          (terminal normal ,nn1 (star (seq (notp (terminal ,nn))
                                           (trivial (anychar)))) (()()))
          (terminal token  ,nn2 (terminal ,nn) (() ()))
          (terminal term   ,nm (star (seq (terminal ,nn1)
                                          (bind ret (terminal ,nn2))))
                                          (() (var ret)))
                                          )
       (function ,nm (str) (reverse (map cdr (pfront:easyparse ,nm str)))))
  }
}
```

The example above shows another useful feature of PFront: a fallback S–expressions syntax. It is possible to use such a syntax after the '`#`' character wherever `expr` is allowed.

S–expressions syntax is fully compatible with the core MBase parser, with one addition — a PFront expression (or even a top–level statement) may follow the `=pf:` prefix, as in this example:

```
#(let ((a =pf: 2+2) (b 3)) =pf: a*a+b*b)
```

The most appropriate and obvious use of this feature is in macros, including the kind of syntax extension support macros discussed above. Of course it is possible to construct MBase code using only the core PFront list constructing syntax, but a native Lisp quasiquotation in many cases is more readable. And of course, fallback syntax should not be abused, misused and overused.

# 10 Not.NET extension

MBase is tightly integrated with .NET. The most basic form of integration is the embedded IL code[3] — a powerful but complicated way of generating a low level code. There is a higher level replacement available — Not.NET embeddable programming language. When used with PFront syntax it looks pretty much like C#.

## 10.1 Gluing up .NET code

Because of PFront's syntax extensibility it was possible to add a language designed specifically for interacting with the .NET world. It is achieved through a special `.net:` quotation construct. The value returned back from quotation is a first class citizen both for MBase and PFront language so it can be used anywhere where a value is expected:

[3] `(n.asm ...)` construction

```
function today() return .net: System.Date.get_Now();
```

Please note that there is no syntax sugar in Not.NET language for C# properties.

To make this bidirectional interaction possible, however, we need to introduce a context, which is a list of bound variables that will be allowed into the Not.NET code block. So a slightly more complex example may look like this:

```
function file_size(fname)
  return .net(fname): {
    file = new System.IO.File((string)fname);
    leave file.get_FileSize();
  }
```

It is worth mentioning that nothing like a "duck-typing"' found in many dynamic languages happens here, the variable types are known in the compilation time, so the appropriate CIL method call can be generated, and the resulting code is that efficient as CIL allows. Internally, a type propagation scheme is used to infer the actual type. It is very similar, in nature, to well known C# 'var' construct, however, with a smaller syntax overhead.

All the variables passed into Not.NET code are considered to be of a System.Object type, unless the exact type is given. If it is a value type, a variable value is unboxed prior to casting.

For example: `notnet(int x, double y) { leave (double)x + Math.Sin(y);}`

One of the obvious uses for Not.NET language is a performance optimisation. The code written in Not.NET is executed as fast as a C# code, whereas a raw MBase can be somewhat slower due to the dynamic typing and boxing.

### 10.1.1 `.net` vs. `notnet`

There are two different constructions in PFront for embedding the Not.Net language. The first one, `.net(...): ...`, is for embedding expressions, and it allows only capturing the `System.Object` type variables. The more heavyweight construction is `notnet(...) { ... }`, which captures variables of arbitrary types and embeds a statement instead of an expression.

One significant difference between C# and Not.NET is that the latter allows a sort of "statement–expressions", or "begin–expressions". These expressions looks like normal code blocks which ends with a `leave` construction. For example:

```
notnet(double a, double b, double len)
 { if({x = a*a; y = b*b; leave (Math.Sqrt(x+y));} < len)
     { leave 1;} else {leave 0;} }
```

Here we can see a code block in a place where an expression syntax is expected. It allows to partially leverage the damage of separating statements from expressions, and it is especially useful for code generation, where otherwise a complicated declarations lifting will be required.

Another special language feature in Not.NET is a class–level declarations explicit lifting. It is the only way to add something to the current class from the Not.NET code embedded into a PFront expression.

For example:

```
.net: {
   lift: static int counter;
   lift: static int inc() {
            this.counter = this.counter + 1;
            return this.counter;
         };
   leave this.inc();
 }
```

### 10.1.2 Not.NET quasiquotation and syntax extensions

Not.NET language is as extensible as PFront itself. There is a special quasiquotation syntax that allows to form Not.NET AST. For example:

```
.net-statement ‘System.Console.WriteLine( \exp\ );‘
```

```
.net-expr ‘Math.Cos( \a\ )+Math.Sin( \b\ )‘
```

The other quasiquotation prefixes are `.net-member` and `.net-class`.

Several syntax extentions entries are available in the default Not.NET parser. It is possible to plug in a new syntax:

```
syntax of pfnotnet in nnstmt, start: ’ [pfexpr]:r "=>" [pfexpr]:l ";" ’
{
 return .net-statement ‘\l\ = \r\;‘
}
```

And from now on, we can use this new syntax in Not.NET code:

```
.net: {2 => x; 3 => y; leave x*y;}
```