OMeta
an OO Language for Pattern Matching

Alessandro Warth and Ian Piumarta
UCLA / Viewpoints Research Institute
Programming language research

idea → prototype → validation

Lexical Analysis
AST Transformations
Parsing
Code Generation
We have special weapons...

**lex**, for lexical analysis  
**yacc**, for parsing  
**visitors**, for AST transformations and code generation
... but prototypes are still “expensive”

what about me?

idea

prototype

validation

what

about

me?
Why do we care?

- PL researchers have lots of ideas...
- ...but can only afford to prototype a few of the “more promising” ones
The ideal prototype

• ... should be
  • quick to implement
  • easy to change
  • extensible
  • “efficient enough”
OMeta

• An Object-Oriented language for Pattern Matching
• Intended for rapid language prototyping (but not limited to that domain)
• OO: extend your prototypes using familiar mechanisms
  • inheritance, overriding, ...
• OMeta compiler
• JavaScript compiler
  • “almost” ECMA-262 compliant
• Some JS extensions
• Toylog interface to Prolog for children
Roadmap

• OMeta’s pattern matching
• Object-Oriented features
• Other interesting features
• Experience
Why Pattern Matching?

It’s a unifying idea!

- **lexical analysis**: characters $\rightarrow$ tokens
- **parsing**: tokens $\rightarrow$ parse trees
- **constant folding and other optimizations**: parse trees $\rightarrow$ parse trees
- **(naive) code generation**: parse trees $\rightarrow$ code
Pattern Matching

• ML-style pattern matching
  • Can you write a lexer / parser with it?
  • Yes, but...
  • “That’s what ML-lex and ML-yacc are for!”
• OMeta is based on PEGs
Parsing Expression Grammars (PEGs) [Ford, ‘04]

- Recognition-based foundation for describing syntax
- Only prioritized choice
  - no ambiguities
  - easy to understand
- Backtracking, unlimited lookahead
- Semantic predicates, e.g., ?[x == y]
About the examples

• 2 versions of OMeta:
  • OMeta/Squeak
  • OMeta/COLA

• Slightly different syntaxes

• Use different languages for semantic actions and predicates
PEGs, OMeta style

dig ::= ("0" | ... | "9"):d  =>  [d digitValue]

num ::= <num>:n <dig>:d   =>  [n * 10 + d]
   | <dig>

expr ::= <expr>:e "+" <num>:n  =>  [{#plus. e. n}]
   | <num>
Increasing Generality

• PEGs operate on streams of characters
• OMeta operates on streams of objects
  • `<anything>` matches any one object
  • characters, e.g., `$x$
  • strings, e.g., `'hello'`
  • numbers, e.g., `42`
  • symbols, e.g., `#answer`
• lists, e.g., `{‘hello’ 42 #answer {}}`
Example: evaluating parse trees

\[
\begin{align*}
\text{num} & ::= \text{<anything>:n} \ ?[n \ \text{isNumber}] \implies [n] \\
\text{eval} & ::= \{\#\text{plus} \ \text{<eval>:x} \ \text{<eval>:y}\} \implies [x + y] \\
& \quad \mid \ <\text{num}>
\end{align*}
\]

\{\#\text{plus.} \ \{\#\text{plus.} \ 1. \ 2\}. \ 3\} \rightarrow 6
OMeta is Object-Oriented

OMeta Base

```
anything ::= ...
...
```

dig ::= ("0" | ... | "9"):d => [d digitValue]

num ::= <num>:n <dig>:d => [n * 10 + d] | <dig>

expr ::= <expr>:e "+" <num>:n => [{#plus. e. n}] | <num>

MyLang

```
```

MyLang++

```
expr ::= <expr>:e "-" <num>:n => [{#minus. e. n}] | <super #expr>
```
Extensible pattern matching

meta NullOptimization {
    opt ::= (OR <opt>*:xs) => `(OR ,@xs)
    | (NOT <opt>:x) => `(NOT ,x)
    | (MANY <opt>:x) => `(MANY ,x)
    | (MANY1 <opt>:x) => `(MANY1 ,x)
    | (define <_:n <opt>:v) => `(define ,n ,v)
    | (AND <opt>*:xs) => `(AND ,@xs)
    | (FORM <opt>*:xs) => `(FORM ,@xs)
    | <_;
}

meta OROptimization <: NullOptimization {
    opt ::= (OR <opt>:x) => x
    | (OR <inside>:xs) => `(OR ,@xs)
    | <super opt>;
    inside ::= (OR <inside>:xs) <inside>:ys => (append xs ys)
    | <super opt>:x <inside>:xs => (cons x xs)
    | <empty> => nil;
}
Parameterized productions

digit ::= "0" | "1" | "2" | "3" | "4"
| "5" | "6" | "7" | "8" | "9"

range :a :b ::= <anything>:x ?[x >= a] ?[x <= b] => [x]
digit ::= <range $0 $9>
More about parameterized productions

- The syntax

  \[\text{range} : a : b ::= \ldots\]

  is really shorthand for

  \[\text{range} ::= <\text{anything}> : a <\text{anything}> : b (\ldots)\]

- Arguments prepended to input stream

- Enables pattern matching on arguments

  \[
  \begin{align*}
  \text{fac} & \quad 0 & \Rightarrow & \quad [1] \\
  \text{fac} & \quad : n ::= <\text{fac} (n - 1)> : m & \Rightarrow & \quad [n \times m]
  \end{align*}
  \]
Higher-order productions

formals ::= <name> ("," <name>)*
args    ::= <expr> ("," <expr>)*

listOf :p ::= <apply p> ("," <apply p>)*

formals ::= <listOf #name>
args      ::= <listOf #expr>
MetaCOLA = OMeta + COLA

- duplicated effort
- versioning problem
Foreign production invocation

• Lend input stream to another grammar

    meta MetaCOLA {
        mcola ::= <foreign OMeta 'ometa>
               | <foreign COLA 'cola>;
    }

• Compose multiple grammars w/o worrying about name clashes
Lexically-scoped syntax extensions

(define puts
  (lambda (s)
    (let ((idx 0))
      (while (!= (char@ s idx) 0)
        (putchar (char@ s idx))
        (set idx (+ idx 1)))
      (putchar 10))))
Lexically-scoped syntax extensions

(define puts
  (lambda (s)
    (let ((idx 0))
      (while (!= s[idx] 0)
        (putchar s[idx])
        (set idx (+ idx 1)))
      (putchar 10))))
Lexically-scoped syntax extensions

(define puts
  (lambda (s)
    (let ((idx 0))
      { cola ::= <cola>:a '][' <cola>:i ']' => `(char@ ,a ,i)
        | <super cola>; }
      (while (!= s[idx] 0)
        (putchar s[idx])
        (set idx (+ idx 1)))
      (putchar 10)))))

(puts "this is a test") ;; works
(printf "%d\n" "abcd"[0]) ;; parse error!
Experience

- The OMeta compiler
  - parser, optimizer passes, codegen
- JS compiler (OMeta/Squeak)
  - ~350 LOC (OMeta) for parser, “declaration visitor”, codegen
  - ~1000 lines of JS for libraries
- “almost” ECMA-262 compliant
Experience (cont’d)

- **MJavaScript** = Javascript + Macro support
- ~40 LOC, including additional syntax and macro expansion pass

```javascript
macro @repeat(numTimes, body) {
    var n = numTimes
    while (n-- > 0)
        body
}

@repeat(10 + 5, alert(“hello”))
```
Experience (cont’d)

• **Toylog** = Prolog front-end for children

• ~70 LOC

Homer is Bart’s father.
Marge is Bart’s mother.
x is y’s parent if x is y’s father
or x is y’s mother.
x is Bart’s parent?
Selected Related Work

- Parsing Expression Grammars [Ford, ‘04]
- LISP70 Pattern Matcher [Tesler et al., ‘73]
- Parser combinator libraries [Hutton, ‘92]
- “Modular Syntax” [Grimm, ’06]
http://www.cs.ucla.edu/~awarth/ometa

<questions>