OMeta: an Object-Oriented Language for Pattern Matching *

Alessandro Warth
Computer Science Department
University of California, Los Angeles
and Viewpoints Research Institute
awarth@cs.ucla.edu

Ian Piumarta
Viewpoints Research Institute
piumarta@speakeasy.net

Abstract

This paper introduces OMeta, a new object-oriented language for pattern matching. OMeta is based on a variant of Parsing Expression Grammars (PEGs) [5]—a recognition-based foundation for describing syntax—which we have extended to handle arbitrary kinds of data. We show that OMeta's general-purpose pattern matching provides a natural and convenient way for programmers to implement tokenizers, parsers, visitors, and tree transformers, all of which can be extended in interesting ways using familiar object-oriented mechanisms. This makes OMeta particularly well-suited as a medium for experimenting with new designs for programming languages and extensions to existing languages.

Categories and Subject Descriptors D.1.5 [*Programming Techniques*]: Object-Oriented Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Design, Languages

Keywords pattern matching, parsing, metacircular implementation

1. Introduction

Many problems in computer science, especially in programming language implementation, involve some form of pattern matching. Lexical analysis, for example, consists of finding patterns in a stream of characters to produce a stream of tokens. Similarly, a parser matches a stream of tokens against a grammar—which itself is a collection of productions, or patterns—to produce parse trees. Several other tasks, such as constant folding and (naïve) code generation, can be implemented by pattern matching on parse trees.

Despite the fact that these are all instances of the same problem, most compiler writers use a different tool or technique (e.g., *lex*, *yacc*, and the *visitor design pattern* [6]) to implement each compilation phase. As a result, the skill of programming language implementation has a steep learning curve (because one must learn how to use a number of different tools) and is not widely understood.

Several popular programming languages—ML, for instance—include support for pattern matching. Unfortunately, while ML-style pattern matching is a great tool for processing structured data, it is not expressive enough on its own to support more complex pattern matching tasks such as lexical analysis and parsing.

Perhaps by providing programming language support for a more general form of pattern matching, many useful techniques such as parsing—a skill more or less exclusive to "programming languages people"—might become part of the skill-set of a much wider audience of programmers. (Consider how many Unix applications could be improved if suddenly their implementors had the ability to process more interesting configuration files!) This is not to say that general-purpose pattern matching is likely to subsume specialized tools such as parser generators; that would be difficult to do, especially in terms of performance. But as we will show with various examples, general-purpose pattern matching provides a natural and convenient way to implement tokenizers, parsers, visitors, and tree transformers, which makes it an unrivaled tool for rapid prototyping.

This work builds on Parsing Expression Grammars (PEGs) [5] (a recognition-based foundation for describing syntax) as a basis for general-purpose pattern matching, and makes the following technical contributions:

1. a generalization of PEGs that can handle arbitrary kinds of data (i.e., not just streams of characters) and supports parameterized and higher-order productions (Section 2),

* This material is based upon work supported by the National Science Foundation under Grant No. 0639876. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

expression	meaning
$e_1\ e_2$	sequencing
e_1 | e_2	*prioritized* choice
e^*	zero or more repetitions
e^+	one or more repetitions (not essential)
$\sim e$	negation
$<p>$	production application
`'x'`	matches the character `x`

Table 1. Inductive definition of the language of *parsing expressions* (assumes that e, e_1, and e_2 are parsing expressions, and that p is a non-terminal).

2. a simple yet powerful extensibility mechanism for PEGs (Section 3),
3. the design and implementation of *OMeta*, a programming language with convenient BNF-like syntax that embodies (1) and (2), and
4. a series of examples that demonstrate how our general-purpose pattern matching facilities may be used in the domain of programming language implementation.

The rest of this paper explores our notion of general-purpose pattern matching in the context of OMeta.

2. OMeta: an extended PEG

An OMeta program is a Parsing Expression Grammar (PEG) that can make use of a number of extensions in order to handle arbitrary kinds of data (PEGs are limited to processing streams of characters). This section begins by introducing the features that OMeta and PEGs have in common, and then describes some of OMeta's extensions to PEGs.

2.1 PEGs, OMeta Style

Parsing Expression Grammars (PEGs) [5] are a recognition-based foundation for describing syntax. A PEG is a collection of productions of the form *non-terminal* → *parsing-expression*; the language of parsing expressions is shown in Table 1.

To avoid ambiguities that arise from using a non-deterministic choice operator (the kind of choice found in CFGs), PEGs only support *prioritized choice*. In other words, choices are always evaluated in order. As a result, there is no such thing as an ambiguous PEG, and their behavior is easy to understand.[1]

Figure 1 shows a PEG, written in OMeta syntax, that *recognizes* simple arithmetic expressions (it does not create parse trees). In order to create parse tree nodes and/or do anything else upon successful matches, the programmer must write *semantic actions*.

[1] Although the use of left-recursive productions should result in infinite recursion (because of prioritized choice), some PEG implementations, including OMeta, provide special support for left recursion as a convenience.

```
meta E {
  dig ::= '0' | ... | '9';
  num ::= <dig>+;
  fac ::= <fac> '*' <num>
        | <fac> '/' <num>
        | <num>;
  exp ::= <exp> '+' <fac>
        | <exp> '-' <fac>
        | <fac>;
}
```

Figure 1. A PEG, written in OMeta, that recognizes simple arithmetic expressions.

Semantic actions are specified using the `=>` operator and written in OMeta's *host language*, which is usually the language in which the OMeta implementation was written. We currently have two implementations: one written in COLA [11], and another in Squeak Smalltalk[2], both of which can be downloaded from `http://www.cs.ucla.edu/~awarth/ometa/`.

Important: All of the examples in this paper were written for the COLA implementation of OMeta; if you are not familiar with COLA, try to think of it as a cross between Scheme and Smalltalk.

The syntax of the functional parts of the language is very similar to Scheme, e.g., `(+ 1 2 (f 5))`.

COLA message sends look similar to Smalltalk's, but are written in square brackets, e.g., `[jimmy eat: banana]`. Note that each `[]`-expression represents a single message send; the COLA translation of the Smalltalk message `Array new: x size` is `[Array new: [x size]]`.

Refer to `http://piumarta.com/pepsi/coke.html` for a more detailed description of COLA syntax.

Here is one way our grammar's `exp` production might be modified in order to create parse trees (the other productions in our grammar would have to be modified accordingly):

```
exp ::= <exp>:x '+' <fac>:y => `(+ ,x ,y)
      | <exp>:x '-' <fac>:y => `(- ,x ,y)
      | <fac>;
```

Note that the results of `<exp>` and `<fac>` are bound to identifiers (with the `:` operator) and referenced by the semantic actions to create parse tree nodes. Note also that the last choice in this production, `<fac>`, does not specify a semantic action. In the absence of a semantic action, the value returned by a production upon a successful match is the result of the last expression evaluated (hence `<fac>` is equivalent to `<fac>:x => x`).

[2] The Squeak port of OMeta was joint work with Yoshiki Ohshima.

OMeta has a single built-in production from which every other production is derived. The name of this production is _ (underscore), and it consumes exactly one element from the input stream. Even the `end` production, which detects the end of the input stream, is implemented in terms of _:

```
end ::= ~<_>;
```

In other words, we are at the end of the input stream if it is not possible to consume another element from it. As noted in [5], the ~ operator—used for negation—can also be used to provide unlimited look-ahead capability. For example, `~~<expr>` ensures that an `expr` follows, but does not consume any input.

Like several other PEG implementations (e.g., Bryan Ford's *Pappy* [3]), OMeta also supports *semantic predicates* [10]: host language (boolean) expressions that can be evaluated while pattern matching. In OMeta, semantic predicates are written using the `?` operator. For example, the following production matches numbers greater than 100:

```
largeNumber ::= <number>:n ?(> n 100) => n;
```

2.2 PEG Extensions for Generality

PEGs operate on streams of characters, and consequently, they only support one kind of *primary* parsing expression: characters. Because OMeta operates on arbitrary kinds of data, it needs to support some additional kinds of expression:

- strings (e.g., `"hello"`)
- numbers (e.g., `42`)
- atoms (e.g., `answer`)
- lists (e.g., `("hello" 42 answer ())`)

Note that the patterns `'x' 'y' 'z'` and `"xyz"` are not equivalent: the former matches three character objects, whereas the latter matches a single string object. On the other hand, the patterns `('x' 'y' 'z')` and `"xyz"` *are* equivalent, because a string can always be viewed as a list of characters. For convenience, OMeta accepts the syntax `'xyz'` (in single quotes) as shorthand for the sequence `'x' 'y' 'z'`.

List patterns enable OMeta grammars to handle arbitrarily-structured data. A list pattern may contain a nested pattern, which itself may be any valid parsing expression (a sequence of patterns, another list pattern, etc.). In order for a list pattern *p* to match a value *v*, two conditions must be met: (i) *v* must be a list, or list-like entity (e.g., a string), and (ii) *p*'s nested pattern must match the contents of *v*. The list pattern `(<_>*)`, for example, matches any list.

Figure 2 shows a simple OMeta grammar that uses list patterns to flatten a list. Feeding the list

```
(1 (2 (3 4)) (((5)) 6))
```

to our grammar's `flatten` production produces the flattened list `(1 2 3 4 5 6)`.

2.3 PEG Extensions for Expressiveness

OMeta's productions, unlike those in PEGs, may take any number of arguments. This feature can be used to implement a lot of functionality that would otherwise have to be built into the language. As an example, consider regular-expression-style *character classes*, which traditional PEG implementations support in order to spare programmers from the tedious and error-prone job of writing productions such as

```
letter ::= 'a' | 'b' | 'c' | ... | 'y' | 'z';
```

and instead allow them to write the more convenient

```
letter ::= [a-z];
```

Using *parameterized productions* (i.e., productions with arguments), OMeta programmers can write

```
cRange x y ::= <char>:c ?(>= c x)
                        ?(<= c y) => c;
```

which is just as convenient to use as character classes (e.g., `<cRange 'a' 'z'>`), and much more flexible because it is completely programmer-accessible.

The combination of parameterized productions and semantic predicates can be used to support a hybrid of (traditional) "scannerful" and *scannerless parsing* [13], as shown below:

```
eq      ::= '='             => '(eq  nil);
num     ::= <digit>+:ds     => `(num ,ds);
id      ::= <letter>+:ls    => `(id  ,ls);

scanner ::= <space>* {<eq> | <num> | <id>};
tok t   ::= <scanner>:x ?(== [x first] t)
                            => [x second];

assign  ::= <tok 'id> <tok 'eq> <tok 'num>;
```

We have found this idiom to be less error-prone than scannerless parsing (the only kind supported by PEGs), and yet just as expressive since each production may access the character stream directly if desired.

OMeta also provides a mechanism for implementing higher-order productions, using the `apply` production. For example, the production

```
listOf p ::= <apply p> {',' <apply p>}*;
```

(where the {}s are used for aggregation) can be used to recognize both lists of expressions (`<listOf 'expr>`) and lists of names (`<listOf 'name>`).

These extensions bring some of the expressive power of parser combinator libraries [8, 9] to the world of PEGs.

```
meta F {
  flatten ::= (<inside>:xs)             => xs;
  inside  ::= (<inside>:xs) <inside>:ys => (append xs ys)
            | <_>:x         <inside>:xs => (cons   x  xs)
            | <empty>                   => nil;
}
```

Figure 2. Flattening lists.

2.4 A Note on Memoization

Packrat parsers are parsers for PEGs that are able to guarantee linear parse times while supporting backtracking and unlimited look-ahead "by saving all intermediate parsing results as they are computed and ensuring that no result is evaluated more than once." [4] While OMeta is based on PEGs, it does not necessarily have to be implemented using packrat-style memoization.

Our COLA-based implementation does in fact memoize the results of productions without arguments, but in order to keep its memory footprint small, we chose not to memoize the results of productions with arguments. Our Squeak-based implementation, on the other hand, does not memoize any results.

While the linear time guarantee that comes with memoization is certainly desirable, some of our experiments with PEGs indicate that the overhead of memoization may outweigh its benefits for the common case, where backtracking is limited. These trade-offs are orthogonal to the ideas discussed in this paper.

3. O is for Object-Oriented

Programming in OMeta would be very frustrating if all productions were defined in the same namespace: two grammars might unknowingly use the same name for two productions that have different purposes, and one of them would certainly stop working! (Picture one sword-wielding grammar decapitating another, *Highlander*-style: "There can be only one!")

A *class* is a special kind of namespace that comes with a huge bonus: a familiar and well-understood extensibility mechanism. By making OMeta an object-oriented language (i.e., making grammars analogous to classes and productions analogous to methods), several interesting things became possible.

3.1 Quick and Easy Language Extensions

Programming language researchers often implement extensions to existing languages in order to experiment with new ideas in a real-world setting. Consider the task of adding a new kind of loop construct to Java, for example; Figure 3 shows how this might be done in OMeta by creating a new parser that inherits from an existing Java parser and overrides the production for parsing statements. Note that the application `<super stmt>` behaves exactly like a super-send in traditional OO languages.

Note: The example above is the only one in this paper that does not actually run in our implementation, the reason being that we do not have a Java parser written in OMeta. What we *do* have is an almost complete implementation of Javascript, which we discuss in Section 5.

3.2 Extensible Pattern Matching

The OMeta parser (the front-end of our implementation) translates the code for a production, which is a stream of characters, into a parse tree. It represents sequences as `AND` nodes, choices as `OR` nodes, applications as `APPLY` nodes, and so on. As an example, the parse tree generated for the body of the production

```
foo ::= <bar> <baz>;
```

is

```
(OR (AND (APPLY bar) (APPLY baz)))
```

which is later transformed by the OMeta compiler into the code that implements that production.

Our simple-minded parser always produces an `OR` node for the body of a production, even when there is only one alternative (as in the example above). This is wasteful, and can degrade the performance of OMeta programs. After all, the `OR` macro must store the current position of the parser's input stream so that when a choice fails, it can backtrack before trying the next choice.

Expressions like

```
(OR (OR (APPLY x) (APPLY y)) (APPLY z))
```

are also needlessly inefficient. `OR`s are associative, and thus the expression above can be flattened to the more efficient

```
(OR (APPLY x) (APPLY y) (APPLY z))
```

Our implementation performs several such transformations in order to improve the performance of OMeta programs. Each of these is implemented in OMeta itself, using an idiom similar to the visitor design pattern. Figure 4 shows (i) the `NullOptimization` grammar, which visits each node in a production's parse tree, and (ii) the `OROptimization` grammar, which inherits the traversal code from `NullOptimization` and overrides the `opt` production in order to implement the two optimizations for `OR` nodes described in this section.

```
meta Java++ <: Java {
  stmt ::= <space>* 'foreveryother' <space>* '(' <expr>:x <space>* ')' <stmt>:s => ...
         | <super stmt>;
}
```

Figure 3. Extending Java with `foreveryother` loops.

We have implemented several other transformations, including a *jumptable*-based optimization that allows choices such as `(OR (CHAR 97) (CHAR 98) (CHAR 99))` to be performed in constant time, and left factoring.[3]

3.3 Stateful Pattern Matching

OMeta's grammars may be declared to have any number of instance variables. These variables are initialized by the `__init__` production, which is invoked automatically when a new instance of the grammar is created.[4]

Using an earlier version of OMeta, we implemented a parser for a significant subset of Python [12] that used an instance variable to hold a stack of indentation levels. This stack was used for implementing Python's *offside rule*, which enables programs to use indentation instead of brackets for forming lexical scopes.

Another example of OMeta's stateful grammars is `Calc`, a desk calculator grammar, shown in Figure 5. This grammar is not just a parser; it is a complete *interpreter* for arithmetic expressions with variables (the interpreting is done in the productions' semantic actions). `Calc`'s instance variable `vars` holds a symbol table that maps variable names to their current values. The following transcript shows our desk calculator in action:

```
> 3+4*5
23
> x = y = 2
2
> x = x * 7
14
> y
2
```

Note that OMeta does not attempt to undo the effects of a production's semantic actions while backtracking (for some semantic actions, like printing characters to the console, this would be impossible). Programmers implementing stateful pattern matchers must therefore write their semantic actions carefully.

[3] These transformations are part of our implementation, available at `http://www.cs.ucla.edu/~awarth/ometa/`.

[4] A grammar object must be instantiated before it can be used to match a value with a start symbol (production). This is done by sending the grammar the `match:with:` message (e.g., `[G match: '(1 2 3) with: 'myList]`).

3.4 Foreign Production Invocation

Consider the task of implementing a parser for MetaCOLA, a language that is the union of COLA and OMeta. (Suppose we already have OMeta parsers for these languages; they are called `COLA` and `OMeta`, respectively.)

Using OMeta's single inheritance mechanism, we could either

1. make `MetaCOLA` inherit from `COLA` and duplicate (re-implement) `OMeta`'s productions, or
2. make `MetaCOLA` inherit from `OMeta` and duplicate `COLA`'s productions,

but neither of these choices is satisfactory, since it results in code bloat and creates a versioning problem (e.g., subsequent changes to the `OMeta` parser will not carry over to the the `MetaCOLA` parser resulting from (1)). Making `MetaCOLA` inherit from both `OMeta` and `COLA` would also be a bad idea, since name clashes would most likely result in incorrect behavior.[5]

A much better solution to this problem is OMeta's *foreign production invocation* mechanism, which allows a grammar to "lend" its input stream to another in order to make use of a foreign production. This mechanism is accessed via the `foreign` production, which takes as arguments the foreign parser and production name, as shown below:

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

Foreign production invocation makes it possible for programmers to compose multiple grammars without having to worry about name clashes.

4. More Examples

4.1 Lexically-Scoped Syntax Extensions

A Domain-Specific Language (DSL) is a programming language designed to do one kind of task very well. OMeta, for instance, can be thought of as a DSL for pattern matching.

While DSLs are used for writing entire programs, Mood-Specific Languages (MSLs) are intended for writing just a few lines of code in order to make *part of a program* easier to write. Our COLA parser, which was written in OMeta, supports this notion by allowing programmers to create *lexically-scoped syntax extensions*.

[5] OMeta does not support multiple inheritance.

```
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;
}
```

Figure 4. Extensible pattern matching in OMeta.

```
meta Calc (vars) {
  __init__ ::= <empty>                             => [self vars: [IdentityDictionary new]];
  space    ::= ' ';
  var      ::= <letter>:x <space>*                 => x;
  num      ::= <num>:n <digit>:d                   => [[n * '10] + [d - '$0]]
             | <digit>:d <space>*                  => [d - '$0];
  priExpr  ::= <var>:x                             => [[self vars] at: x]
             | <num>:n                             => n
             | '(' <space>* <expr>:r ')' <space>*  => r;
  mulExpr  ::= <mulExpr>:x '*' <space>* <priExpr>:y => [x * y]
             | <mulExpr>:x '/' <space>* <priExpr>:y => [x / y]
             | <priExpr>;
  addExpr  ::= <addExpr>:x '+' <space>* <mulExpr>:y => [x + y]
             | <addExpr>:x '-' <space>* <mulExpr>:y => [x - y]
             | <mulExpr>;
  expr     ::= <var>:x     '=' <space>* <expr>:r    => [[self vars] at: x put: r]
             | <addExpr>;
  rep      ::= <space>* <expr>:r '\n'              => (println r);
}
```

Figure 5. A desk calculator.

Programmers may create syntax extensions inside any COLA s-expression by writing one or more OMeta productions inside {}s. This creates a new parser object (at parsing time) that inherits from the current parser, giving the programmer a chance to override the original parser's productions in order to extend the language as desired. The language extension is in scope until the end of the current s-expression.

Figure 6 shows a COLA program that implements the `puts` function (which, like the C version, prints a string followed by the newline character to the console). In order to make array indexing operations more convenient, `puts` locally extends COLA with syntax for C-style array indexing. This enables the programmer to write `s[i]` where he would otherwise have to write the less readable `(char@ s i)`. Note that the expression `(printf "%d\n" "abcd"[0])`, which appears outside the scope of this syntax extension, results in a parse error.

Extending our COLA parser with support for lexically-scoped syntax extensions was straightforward:

```
(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!
```

Figure 6. A lexically-scoped syntax extension for C-style array accesses.

- the productions of the syntax extension are parsed using foreign production invocation (this is possible because the OMeta parser itself was implemented in OMeta), and
- compiling and instantiating a new parser object at parsing time was an easy task, since OMeta is a dynamic language.

This feature can just as easily be added to any other language whose parser is implemented in OMeta (the Javascript implementation described in the next section, for example).

4.2 Checking `printf` Calls

In order to give programmers access to operating system libraries, COLA programs are able to call C functions directly. This is of course a very low-level operation, and can result in crashes and security problems. The `printf` function, whose expected number of arguments depends on a *format string*, is notorious for making programs unsafe.[6]

Modern C compilers are aware of this problem and implement static checks in order to make sure that `printf` is given the correct number of arguments. The COLA compiler, on the other hand, does not give `printf` any special treatment. Consequently, calling `printf` from a C program is somewhat safer than calling it from a COLA program.

Figure 7 shows an OMeta grammar that processes an entire COLA program, replacing "bad" `printf` calls (those whose number of arguments is not in agreement with the format string) with the atom `BAD_PRINTF`. For example, it transforms the COLA program

```
(begin
  (printf "my name is %s\n" name)
  (if (> (printf "%s = (%d,%d) %f" name x y)
         5)
      (f x)))
```

into

```
(begin
  (printf "my name is %s\n" name)
  (if (> BAD_PRINTF 5)
      (f x)))
```

Here is a brief explanation of each production in the grammar shown in Figure 7:

- `rep` (repeat) is a higher-order production that takes two arguments, an integer `n` and a production `p`, and tries to apply production `p` `n` times.
- `fmt` traverses a format string, and returns the number of arguments it expects.
- `expr` visits an entire COLA program (an s-expression). When it comes across a call to `printf`, it finds out how many arguments are required by the format string and uses `rep` to ensure that the call is well-formed. Otherwise, the "bad" call is replaced by the atom `BAD_PRINTF`. Note that the format string is matched against the pattern `(<fmt>:n)`. Any string can be treated as a list; the pattern above means "match a list-like value whose contents are described by `<fmt>`".

This example shows that OMeta's seemingly orthogonal parsing and traditional pattern matching capabilities (used here to process format strings and COLA programs, respectively) can be combined in useful ways.

5. Case Study: Javascript

We have used OMeta to implement a nearly complete subset of the Javascript [2] programming language.[7] Our implementation consists of a parser written in OMeta, and a code generation phase implemented using a combination of OMeta pattern matching and Scheme-style macros. It is currently being used by Dan Ingalls' research group at Sun Labs to run moderately-sided programs (∼2k lines of code).

[6] The same is true for other functions in the `printf` family, such as `sprintf` and `fprintf`.

[7] The missing features are `with` and `try/catch`.

```
meta P {
  rep n p ::= ?[n = '0]
            | <apply p> <rep [n - '1] p>;
  fmt     ::= '%'    '%'    <fmt>
            | '%'    <char> <fmt>:n        => [n + '1]
            | <char>        <fmt>
            | <empty>                      => '0;
  expr    ::= (printf (<fmt>:n) <rep n '_>)
            | (printf <_>*)                => 'BAD_PRINTF
            | (<expr>*:xs)                 => `(,xs)
            | <_>;
}
```

Figure 7. Hunting for bad `printf` calls.

5.1 Parsing

Our Javascript parser is implemented as a 177-line OMeta grammar. An interesting part of this grammar is its mechanism for handling Javascript's *automatic semicolon insertion rules* [2]:

```
sc ::= {~'\n' <space>}* {';'   |
                         '\n'  |
                         ~~'}' |
                         <end> };
```

({}s are used for aggregation.) The `sc` production consumes a (logical) semicolon, which may be one of the following:

- a semicolon character
- a newline character
- nothing at all—but only if the next token is } (double negation is used for look-ahead)
- the end of the input stream

In order to support automatic semicolon insertion, other productions in our parser use `sc` instead of `';'` directly, as shown in `stmt` below:

```
stmt ::= 'continue' <sc> => '(continue)
       | 'break'    <sc> => '(break)
       | ...;
```

5.2 Code Generation

Our Javascript parser represents parse trees as COLA s-expressions. These parse trees are expanded into executable COLA expressions using Scheme-like macros.

Before the macro expansion step, parse trees are processed by an OMeta grammar that distinguishes local variables from non-local variables so that they can be compiled differently (this is necessary to implement the correct semantics).

6. Related Work

Our work on OMeta began when we implemented our own version of Val Schorre's META-II [14]: a simple yet practical recognition-based compiler-writing language that could be implemented in itself in roughly a page of code. META-II was a wonderful tool, but it had significant limitations:

1. it did not support backtracking, which made it necessary for the programmer to do a large amount of left-factoring in productions, and
2. its analog of semantic actions were `PRINT` commands, which meant that compilers had to generate code *while* recognizing programs. (The resulting programs were usually interpreted by a special-purpose virtual machine which had been implemented specially for the language being compiled.)

Once we added backtracking and semantic actions to our language, it became equivalent in power to Bryan Ford's Parsing Expression Grammars (PEGs) [5]. This PEG foundation relates OMeta to packrat parser generators [4] (e.g., Ford's own `Pappy` [3] and Robert Grimm's *Rats!* [7]). However, OMeta is not a parser generator; it is a programming language whose *control structure* is based on PEGs. And unlike previous PEG-based systems—which operate only on streams of characters—our extensions to PEGs (see Section 2) enable OMeta to handle arbitrary kinds of data. Also, packrat parsers are so called because they memoize all intermediate results. Although the COLA port of OMeta does memoize intermediate results, our Squeak port does not. (Some of our experiments with PEGs indicate that the overhead of memoization may outweigh its benefits for the common case, where backtracking is limited.)

OMeta's ability to pattern match over arbitrary kinds of data and the notion of productions with arguments were inspired by the LISP70 system [16]. LISP70 used pattern matching for general programming tasks as well as extending its own syntax. Unlike OMeta, LISP70 used an external lexical analyzer, and was not object-oriented.

OMeta is also related to Haskell's parser combinator libraries [8, 9] in that it supports parameterized and higher-order productions. However, OMeta grammars are more readable than those written using parser combinator libraries, and more extensible due to the object-oriented features discussed in Section 3.

7. Conclusions and Future Work

We have shown that OMeta's pattern matching is a great tool for implementing various tasks in the domain of programming language implementation (e.g., lexical analyzers, parsers, visitors, etc.). This makes OMeta particularly well-suited as a medium for experimenting with new designs for programming languages and extensions to existing languages.

Although our initial implementation of OMeta was written in COLA [11], OMeta can also be implemented using more conventional languages. With Yoshiki Ohshima, for example, we implemented a port of OMeta in Squeak Smalltalk (where OMeta is being used to experiment with alternative syntaxes for the Squeak EToys system). Ports to other languages like Scheme [1] and Common LISP [15] should be relatively straightforward.

While OMeta's parameterized productions provide great expressive power, it is unfortunate that our language does not allow a production's arguments to be pattern-matched against in the production's body. For example, it would be nice to be able to write

```
p ::= 0       ...
    | <_>:n ...;
```

instead of

```
p n ::= ?(== n 0) ...
      |           ...;
```

but the former is not possible because OMeta's production arguments are passed on the stack, whereas all pattern matching is done on the input stream. LISP70 had a better mechanism for argument passing that consisted of inserting the arguments of a production application at the beginning of the input stream (that they could be matched against). We plan to adopt this mechanism in OMeta.

We also plan to improve the performance of our OMeta implementations; it should be possible for them to perform competitively with state-of-the-art packrat parser implementations such as Robert Grimm's *Rats!* [7].

8. Acknowledgments

The authors would like to thank Alan Kay for inspiring this project and providing valuable insights. We would also like to thank Yoshiki Ohshima for making the Squeak port of OMeta possible, and Todd Millstein, Benjamin Titzer, Stephen Murrell, Takashi Yamamiya, Paul Eggert, Jamie Douglass, Tom Bergan, and Mike Mammarella for useful comments on this work.

References

[1] H. Abelson, R. K. Dybvig, C. T. Haynes, G. J. Rozas, I. N. I. Adams, D. P. Friedman, E. Kohlbecker, G. L. Steele, D. H. Bartley, R. Halstead, D. Oxley, G. J. Sussman, G. Brooks, C. Hanson, K. M. Pitman, and M. Wand. Revised report on the algorithmic language Scheme. *SIGPLAN Lisp Pointers*, IV(3):1–55, 1991.

[2] ECMA. ECMAScript language specification.

[3] B. Ford. Pappy: a parser generator for Haskell. `http://pdos.csail.mit.edu/~baford/packrat/thesis/`.

[4] B. Ford. Packrat parsing: simple, powerful, lazy, linear time, functional pearl. In *ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 36–47, New York, NY, USA, 2002. ACM Press.

[5] B. Ford. Parsing Expression Grammars: a recognition-based syntactic foundation. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 111–122, New York, NY, USA, 2004. ACM Press.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns.* Addison-Wesley Professional, January 1995.

[7] R. Grimm. Better extensibility through modular syntax. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 38–51, New York, NY, USA, 2006. ACM Press.

[8] G. Hutton and E. Meijer. Monadic parsing in Haskell. *J. Funct. Program.*, 8(4):437–444, 1998.

[9] D. Leijen and E. Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report Technical Report UU-CS-2001-35, Universiteit Utrecht, 2001.

[10] T. J. Parr and R. W. Quong. Adding semantic and syntactic predicates to LL(k): pred-LL(k). In *Computational Complexity*, pages 263–277, 1994.

[11] I. Piumarta. Open, extensible programming systems. Keynote talk, *Dynamic Languages Symposium*, 2006.

[12] G. Rossum. Python reference manual. Technical report, Centre for Mathematics and Computer Science, Amsterdam, The Netherlands, 1995.

[13] D. J. Salomon and G. V. Cormack. Scannerless NSLR(1) parsing of programming languages. In *PLDI '89: Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pages 170–178, New York, NY, USA, 1989. ACM Press.

[14] D. V. Schorre. META-II: a syntax-oriented compiler writing language. In *Proceedings of the 1964 19th ACM national conference*, pages 41.301–41.3011, New York, NY, USA, 1964. ACM Press.

[15] G. L. Steele. An overview of Common Lisp. In *LFP '82: Proceedings of the 1982 ACM symposium on LISP and functional programming*, pages 98–107, New York, NY, USA, 1982. ACM Press.

[16] L. G. Tesler, H. J. Enea, and D. C. Smith. The LISP70 pattern matching system. In *Proc. of the 3rd IJCAI*, pages 671–676, Stanford, MA, 1973.