Semantic Prototyping in M-LISP: A Representation Independent Dialect of LISP with Reduction Semantics*
(Extended Abstract)

Robert Muller
Aiken Computation Laboratory
Harvard University
email: muller@harvard.edu

Abstract

In this paper we describe a new semantic metalanguage which simplifies prototyping of programming languages. The system integrates Paulson's semantic grammars within a new dialect of LISP, M-LISP, which has somewhat closer connections to the λ-calculus than other LISP dialects such as Scheme. The semantic grammars are expressed as attribute grammars. The generated parsers are M-LISP functions that can return denotational (i.e., higher-order) representations of abstract syntax. We illustrate the system with several examples and compare it to related systems.

1 Introduction

The LISP dialect Scheme [SS75, RE86] has been used by a number of researchers as a semantic metalanguage for the automatic generation of compilers from syntactic and semantic specifications. Wand's Semantic Prototyping System (SPS) [Wan84] uses Scheme as a semantic object language. The system has an ad hoc connection to the YACC parser generator to construct parsers and a syntax directed translation scheme is defined which maps initial representations to Scheme procedures. This corresponds to a denotational semantics. The object program is then interpreted by a virtual Scheme machine.

In this paper we introduce a system which integrates these syntactic and semantic specifications into a unified framework. The system is embedded in a new dialect of LISP, M-LISP [Mul89, Mul90a, Mul90b], which was developed with an emphasis on the kind of operational semantics advocated by Plotkin [Plo81]. M-LISP is closely related to Scheme in that it features dynamic typing, lexical scoping, imperative control features and pairs and functions as principal data types. Unlike Scheme, however, M-LISP is independent of any

*This paper appeared as Harvard University, CRCT Technical Report TR-05-90.

representation of its programs. Since it is independent of McCarthy's original representation scheme for Meta-expressions [McC60], M-LISP has no *quotation* form or any of its related forms *backquote*, *unquote*, or *unquote-splicing* and its programs are not understood to be represented as S-expressions. M-LISP's essential abstract syntax is:

$$M ::= X \mid [] \mid [M\ .\ M] \mid x \mid (M\ M) \mid (\lambda x.M)$$

where X denotes the set of upper-case symbols, M-LISP's symbolic constants, $[M\ .\ M]$ denotes a pair, and x denotes the set of lower-case symbols, M-LISP's identifiers. Like the λ-calculus, M-LISP is fully curried. Proper lists and functions of several arguments may be expressed in concrete syntax:

$$\begin{aligned}
[M_1\ M_2\ \ldots\ M_n] &\equiv [M_1\ .\ [M_2\ .\ [\ \ldots\ [M_n\ .\ []]\ \ldots\ ]]] \\
(\lambda x_1 x_2\ \ldots\ x_n.M) &\equiv (\lambda x_1.(\lambda x_2.(\ \ldots\ (\lambda x_n.M)\ \ldots\ ))) \\
(M_1\ M_2\ M_3\ \ldots\ ) &\equiv ((M_1\ M_2)\ M_3\ \ldots\ )
\end{aligned}$$

Apart from the currying, the denotational semantics of M-LISP is similar to the denotational semantics for Scheme.

Our initial motivation for the facility described here was to compensate for the fact that, unlike S-expression LISP dialects, M-LISP's expression reader is not a parser for M-LISP programs. This property of S-expression LISP has been exploited since the earliest days of LISP, and is in part responsible for the proliferation of LISP dialects. Two drawbacks of this use of the reader are that it is restricted to languages that conform to S-expression syntax, and secondly that it determines representation by notation. We wish to avoid these in our system.

Our solution is to integrate parser generation into M-LISP. This provides a unified framework for syntactic and semantic processing. M-LISP's parser generator is accessed through a special form:

```
(MAKE-PARSER [grammar-rule representation] ... )
```

`MAKE-PARSER` is of type:

$$\texttt{MAKE-PARSER} : \mathit{Attribute\ Grammar} \rightarrow \mathit{Parser}$$

where an attribute grammar is a sequence of productions:

$$\mathit{Attribute\text{-}Grammar} : (\mathit{Grammar\text{-}rule} \times \mathit{Representation})^*$$

and the parser returned by `MAKE-PARSER` is thought of as having the type:

$$\mathit{Parser} : \mathit{String} \rightarrow \mathit{Representation}$$

Intuitively, the grammar rule specifies the structure of a sentence in the language of a syntactic category and the corresponding representation is a specification of a *value*: the

```
(DEFINE parser1
  (MAKE-PARSER
    [(e ::= e1 ``+'' e2)    [PLUS  e1 e2]]
    [(e ::= e1 ``*'' e2)    [TIMES e1 e2]]
    [(e ::= ``('' e1 ``)'') e1]
    [(e ::= integer)        [INT integer]]
    [(e ::= identifier)     [IDENT identifier]]))

(DEFINE type car)
(DEFINE left cadr)
(DEFINE right caddr)
(DEFINE value cadr)
(DEFINE ident cadr)
```

Figure 1: An Expression Language and its List Representation

```
(DEFINE evaluate
  (LAMBDA exp env .
    (CASE (type exp)
      [INT   (value exp)]
      [IDENT (env (ident exp))]
      [PLUS  (+ (evaluate (left exp) env) (evaluate (right exp) env))]
      [TIMES (* (evaluate (left exp) env) (evaluate (right exp) env))])))
```

Figure 2: An Evaluator for the Expression Language

abstract representation of that sentence in M-LISP. There are no restrictions on the types of values. The parser returned by `MAKE-PARSER`, on reading a sentence in the language described by the grammar rule returns the corresponding abstract representation.

The details of the specification language are be filled in in Sections 2 and 3. In Section 4 we compare this facility with related work. Section 5 contains conclusions and sketches future lines of research. See [Mul89, Mul90a] for a more complete exposition of M-LISP and its semantics.

As a preliminary example, consider the simple expression language of Figures 1 and 2.

Figure 1 defines the syntax, the representations of syntax and the selectors for the representations. The language consists of integers and variables combined with addition and multiplication. The values of the variables are determined by an environment. A call of the parser reads a string from the input channel and returns the string's list representation as its value.

```
> (parser1)
```

```
3 * (4 + 2)

[TIMES [INT 3] [PLUS [INT 4] [INT 2]]]
```

Figure 2 contains the valuation function which defines the semantics. The evaluator uses a `CASE` macro to index on the type of the expression.[1] A call of the evaluator returns the meaning of the sentence.

```
> (evaluate (parser1) initial-env)
3 * (4 + 2)

18
```

2 Syntax Specification

The grammar specification language, a restricted BNF, is described informally. A grammar is a sequence of productions:

$$\textit{grammar} ::= \textit{production}^*$$

A production is a structure of the general form:

$$\textit{production} ::= \text{“[”}\ \textit{grammar-rule representation}\ \text{“]”}$$

A *grammar-rule* is specified as:

$$\textit{grammar-rule} ::= \text{“(”}\ \textit{variable}\ \text{“::=”}\ \ \textit{sentence-form}\ \text{“)”}$$

The topmost defined variable is taken as the start symbol of the grammar. This defines the language for which the generated parser is specialized. A *sentence-form* is a sequence of *strings* and *variables*. Strings are enclosed in double quotes ‘‘...’’ and represent absolute lexical entities in the sentence structure. Variables are contiguous sequences of lowercase symbols possibly followed by an integer index. *Lexical variables* represent *micro-syntactic* categories such as integers and strings. There are a few predefined lexical categories: *integer*, *string*, *identifier*, *lowersymbol*, *uppersymbol*, *empty* and *nowhitespace*. This set can be extended through the `MAKE-SCANNER` form which we will not consider here. For example, the grammar rule

```
(s ::= integer ‘‘INTEGER’’ string)
```

defines a sentence which consists of any integer followed by the string ‘‘`INTEGER`’’ followed by any string. When no ambiguity will arise, the quote marks around strings may be omitted. For example, the previous rule may be specified as

[1] In M-LISP macro calls are distinguished from function applications by symbolic constants. M-LISP macros may use brackets as well as parentheses for grouping.

```
(DEFINE parser2
  (MAKE-PARSER
    [(e   ::= uppersymbol)        [SYMBOL uppersymbol]]
    [(e   ::= [])                 [NIL]]
    [(e   ::= [e1 . e2])          [PAIR e1 e2]]
    [(e   ::= id)                 id]
    [(e   ::= (e1 e2))            [APPLICATION e1 e2]]
    [(e   ::= (LAMBDA id . e1))   [PROCEDURE id e1]]
    [(id  ::= lowersymbol)        [IDENT lowersymbol]]))
```

Figure 3: The Standard Representation of M-LISP

```
(s ::= integer1 INTEGER string)
```

Syntactic variables represent syntactic categories. For example, the grammar rule

```
(s ::= ``a'' s ``b'')
```

defines an infinite sequence of "a"s followed by an infinite sequence of "b"s. When two or more occurrences of the same variable appear on the right-hand side of a grammar rule they are indexed. So in

```
(s ::= integer1 INTEGER integer2)
```

the index allows us to distinguish the lexical entity on the left from the lexical entity on the right. Further details of the specification language can be found in [Mul89].

3 Representation

The representation component of a production is a specification of an M-LISP value that the generated parser will use as a template for constructing representations of input sentences. Consider, for example, the parser defined in Figure 3 which maps the abstract syntax of M-LISP to its *standard representation*. Consider the production for applications:

```
[(e   ::= (e1 e2))            [APPLICATION e1 e2]]
```

The grammar rule component describes a sentence composed of a the string ``('' followed by two recursive instances of the `e` category followed by the string ``)''. The representation component specifies that any such sentence will be represented by a list value tagged with the symbol `APPLICATION`. The remaining components are the respective representations of the `e` subexpressions. When the parser is called and presented an input string, it returns this abstract representation. For example,

```
(DEFINE parser3
  (MAKE-PARSER
    [(e ::= e1 ``+'' e2)    (LAMBDA env . (+ (e1 env) (e2 env)))]
    [(e ::= e1 ``*'' e2)    (LAMBDA env . (* (e1 env) (e2 env)))]
    [(e ::= ``('' e1 ``)'') e1]
    [(e ::= identifier)     (LAMBDA env . (env identifier))]
    [(e ::= integer)        (LAMBDA env . integer)]))
```

Figure 4: A Semantic Grammar for the Expression Language

```
> (parser2)
(f x)

[APPLICATION [IDENT F] [IDENT X]]
```

If the representation component `rep`, of a production has, from left to right, grammar variables `v1,...,vn`, then the representation obtained from a call of the parser is equivalent to the value of the application:

```
((LAMBDA v1 ... vn . rep) v1-rep ... vn-rep)
```

Thus, the occurrences of `e1` and `e2` in the representation component of the preceding example may be taken as lambda variables in the context:

```
((LAMBDA e1 e2 . [APPLICATION e1 e2]) e1-rep e2-rep)
```

3.1 Higher-Order Representation of Abstract Syntax

Since functions are values in M-LISP and the representation component of a production provides for the specification of any value type, we may consider higher-order representations of abstract syntax. For example, in Figure 4 we define another parser for the expression language of Figure 1. The *representations* generated by this latter parser embody the semantic processing which is expressed in the *evaluate* function of Figure 2. The new evaluator is given by

```
(DEFINE evaluate
  (LAMBDA exp env . (exp env)))
```

Since M-LISP is fully curried, the expression plus input

```
> (evaluate (parser3))

x * (y + z)
```

denotes a function from environments to values.

3.1.1 Lexically Closed Higher-Order Attribute Grammars

When a functional representation is specified in a call to `MAKE-PARSER`, a call of the generated parser dynamically introduces a function in the run-time context. This raises a question about the resolution of bindings of free variables in the specification component of the production. Given that the parser may be defined in one naming context and called in another there are two obvious choices for the resolution of its free variables.

We have noted that free variables that are grammar variables (i.e., either lexical or syntactic variables) are treated as bound lambda variables. But the representation component may contain free lambda variables. For example,

```
(MAKE-PARSER
  [(s ::= A s1) (LAMBDA x . [s1 x y])])
```

contains a free lambda variable `y`. According to our convention this is logically equivalent to

```
((LAMBDA s1 x . [s1 x y]) s1-rep)
```

When provided input this reduces to the term

```
(LAMBDA x . [s1-rep x y])
```

in which `y` is still free. We specify that the representations are closed in the context of their *definition*. This effectively treats these *specifications* of procedures as though they were lexically scoped procedures. It is safe to ignore this subtle distinction.

Closing grammars lexically ensures that the operators the grammar writer specified in the representation are the operators that are actually used when the abstract syntax is active. Consider, for example, the semantic grammar for M-LISP defined in Figure 5. The representation of abstract syntax corresponds to a standard continuation passing semantics. Thus, the abstract syntax evaluates itself when applied to an environment and a continuation:

```
> ((parser4) initial-env initial-cont)
```

Since the parser is defined within the scope of binding occurrences of `lookup`, `extend` and `close`, these definitions will be available when the abstract syntax is active.

The advantages of integrating compiler compiler functionality into the semantic metalanguage are seen by comparing the semantic grammar of Figure 5 with the comparable SPS definition excerpted from [Wan84] in Figure 6.

Since M-LISP's attribute grammars disengage notation from representation we can define parsers for languages of any syntactic structure such as C, ML or Miranda, and map their syntax to M-LISP values. If we use semantic grammars of the kind illustrated in Figure 5, then the system constitutes a *denotational* prototyping system. M-LISP's close relation to the untyped λ-calculus, the commonly accepted metalanguage of denotational semantics, certifies its aptness as a semantic metalanguage.

```
(DEFINE parser4
  (LET ([lookup (LAMBDA id r . (r id))]
        [extend (LAMBDA r id1 val id2 .
                  (IF (eq? id1 id2) val (r id2)))]
        [close  (LAMBDA id body r act k .
                  (body (extend r id act) k))])
    (MAKE-PARSER
      [(e ::= uppersymbol)      (LAMBDA r k . (k uppersymbol))]
      [(e ::= [])               (LAMBDA r k . (k []))]
      [(e ::= [e1 . e2])        (LAMBDA r k .
                                  (e1 r (LAMBDA fst .
                                          (e2 r (LAMBDA snd .
                                                  (k [fst . snd])))))))]
      [(e ::= id)               (LAMBDA r k . (k (lookup id r)))]
      [(e ::= (LAMBDA id . e1)) (LAMBDA r k . (k (close id e1 r)))]
      [(e ::= (e1 e2))          (LAMBDA r k .
                                  (e1 r (LAMBDA f .
                                          (e2 r (LAMBDA a . (f a k))))))))]
      [(id ::= lowersymbol)     lowersymbol])))
```

Figure 5: A Semantic Grammar for M-LISP

```
(define sdt
  '((yacc-parser 'example.0)              ; the parser;
    (pgm                                  ; a production name
       (pgm exp)                          ; the production: pgm -> exp .
        (e1)                              ; dummy vble for value of exp
        ((e1 init-env init-cont)          ; the value of the node.
         init-state))
    (identifier-to-exp
       (exp ident)                        ; the production: exp -> ident
       (id)
       (lambda (r k) (k (r id))))
    (ident
       (ident literal)                    ; ident -> literal
       (id)
       id)
    ; etc ....
    (appl
       (exp 'lparen exp exp 'rparen)      ; exp -> (exp exp)
       (e0 e1)
       (lambda (r k)
         (e0 r
             (lambda (v0)
               (casefn v0
                 (lambda (n) (error (list 'cant-apply-number)))
                 (lambda (f)
                   (e1 r (lambda (v1) (f v1 k)))))))))))
```

Figure 6: An SPS Specification of a Syntax Directed Transducer

4 Related Work

We have emphasized the connections between M-LISP's parser generator and LISP's *reader* and Wand's semantic prototyping system. A number of other related systems have been proposed in the context of semantics directed compilation. Although this latter issue has not been our subject *per se*, the issues clearly overlap. Mosses' Semantic Implementation System [Mos79] automatically generates a compiler from a denotational specification. The idea of a *semantic grammar* was introduced later by Paulson [Pau81]. Lee and Pleban [LP86, LP87] developed a LISP based compiler generator which relies on modular specification of semantics. Like Nielson's two-level semantics [NN86], Lee and Pleban's model distinguishes between compile-time and run-time aspects of the semantics. Like SPS, Scheme is used as an interpreter for the dynamic semantics.

In [PE88] Pfenning and Elliott describe the role of higher-order abstract syntax in a language-generic environment for formal program development. Their metalanguage, which was developed independently of our model, is a simply typed λ-calculus enriched with pairs and polymorphism. Since our facility is embedded in LISP, it, like the λ-calculus, is an *untyped* embedding language. A Second difference is that higher-order representation of abstract syntax is not *understood* in M-LISP. Rather, since functions are values, it is a representation option for the grammar writer. In some contexts other representations may be more appropriate.

5 Conclusions

We have developed a framework for integrating syntactic processing in a dialect of LISP which, because of its relation to the λ-calculus, is particularly well suited as a semantic metalanguage. The resulting system is an excellent environment for semantic prototyping of programming languages. Designers can enter BNF notation along with the denotational specification and obtain an M-LISP based interpreter. We expect this system to serve as a good framework for semantics directed compilation as well although we have not yet fully investigated this aspect of the system.

5.1 Acknowledgements

We wish to thank Tom Cheatham, A. J. Kfoury and Mitchell Wand for their support and guidance in the development of M-LISP. This research was supported in part by Defense Advanced Research Projects Agency N00039-88-C-0163.

References

[LP86] P. Lee and U. Pleban. On the use of LISP in implementing denotational semantics. In *Proceedings of the ACM Symposium on LISP and Functional Programming*, pages 233–248, 1986.

[LP87] P. Lee and U. Pleban. A realistic compiler generator based on high-level semantics. In *Proceedings of the Fourteenth ACM Symposium on Principles of Programming Languages*, pages 284–295, 1987.

[McC60] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, pages 184–195, 1960.

[Mos79] P. Mosses. Sis — semantics implementation system. Technical Report DAIMI MD-30, Aarhus University, 1979.

[Mul89] R. Muller. *M-LISP: A Representation Independent Dialect of LISP with Reduction Semantics*. PhD thesis, Boston University, 1989.

[Mul90a] R. Muller. M-LISP: A representation independent dialect of LISP with reduction semantics. Technical Report CRCT TR-03-90, Harvard University, 1990.

[Mul90b] R. Muller. Syntax macros in M-LISP: A representation independent dialect of LISP with reduction semantics. Technical Report CRCT TR-04-90, Harvard University, 1990.

[NN86] H. Nielson and F. Nielson. Semantics directed compiling for functional languages. In *Proceedings of the ACM Symposium on LISP and Functional Programming*, pages 249–257, 1986.

[Pau81] L. Paulson. *A Compiler Generator for Semantic Grammars*. PhD thesis, Stanford University, 1981.

[PE88] F. Pfenning and C. Elliott. Higher-order abstract syntax. In *SIGPLAN 88 Conference on Programming Language Design and Implementation*, pages 199–208, 1988.

[Plo81] G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, Denmark, 1981.

[RE86] J. Rees and W. Clinger (Editors). Revised^3 report on the algorithmic language Scheme. *SIGLPAN Notices*, **21, No. 12**:37–79, 1986.

[SS75] G. Steele and G. Sussman. SCHEME an interpreter for extended lambda calculus. Technical Report AI Memo No. 349, Massachusetts Institute of Technology, 1975.

[Wan84] M. Wand. A semantic prototyping system. In *Conference Record of the ACM Conference on Compiler Construction*, pages 166–173, 1984.